



Con Clase, <http://www.conclase.net>

Curso de **MySQL** por Salvador Pozo Coronado

© Mayo 2005

# Prólogo



*MySQL es una marca registrada por MySQL AB. Parte del material que se expone aquí, concretamente las referencias de funciones del API de MySQL y de la sintaxis de SQL, son traducciones del manual original de MySQL que se puede encontrar en inglés en [www.mysql.com](http://www.mysql.com).*

*Este sitio está desarrollado exclusivamente por los componentes de **Con Clase**, cualquier error en la presente documentación es sólo culpa nuestra.*

## Introducción

Siguiendo con la norma de la página de usar software libre, afrontamos un nuevo reto: trabajar con bases de datos mediante el lenguaje de consulta SQL. Este curso será la base para otros que nos permitirán usar bases de datos desde aplicaciones C/C++, PHP, etc.

Originalmente, este curso iba a tratar sólo sobre **MySQL**. Mi intención era limitarlo exclusivamente a explicar la sintaxis de las sentencias y funciones SQL, hacer algunos ejemplos y completar una referencia de **MySQL**.

Sin embargo, como me suele ocurrir cada vez que afronto un nuevo proyecto, las cosas no salen como las tenía planeadas. Poco a poco he ido añadiendo nuevos contenidos, (bastante lógicos, teniendo en cuenta el tema que nos ocupa), y estos contenidos han precisado la inclusión de otros...

Finalmente el curso se ha convertido en algo mucho más extenso, y sobre todo, mucho más teórico, aunque espero que también, en algo mucho más útil.

El curso permitirá (si he sido capaz de explicar todos los conceptos claramente) diseñar bases de datos a partir de problemas reales, haciendo uso de una base teórica firme.

El nivel será, teniendo en cuenta la complejidad del tema de las bases de datos, y el de **MySQL**, bastante básico. Este documento no pretende ser demasiado académico, está orientado a programadores autodidactas que quieran incluir bases de datos en sus aplicaciones. Tampoco entraremos en demasiados detalles sobre configuración de **MySQL**, o sobre relaciones con Apache o Windows. La principal intención es poder manejar bases de datos complejas y grandes, y sobre todo, poder usarlas desde otros lenguajes como C, C++ o PHP.

El presente curso nos obliga a tratar varios temas diferentes.

- Fundamentos teóricos de bases de datos: modelos conceptuales, como el de Entidad-Relación, y modelos lógicos, como el modelo relacional, y herramientas relacionadas con ese modelo, como la normalización.
- Trabajo con servidores. En el caso de **MySQL**, el servidor es el que realiza todas las operaciones sobre las bases de datos, en realidad se comporta como un interfaz entre las bases de datos y nuestras aplicaciones. Nuestras aplicaciones se comunicarán con el servidor para leer o actualizar las bases de datos.
- Por otra parte, trataremos con un lenguaje de consulta y mantenimiento de bases de datos: SQL (Structured Query Language). SQL es un lenguaje en sí mismo, pero mediante el API adecuado podemos usarlo dentro de nuestros propios programas escritos en otros lenguajes, como C o C++.

La teoría sobre bases de datos así como el lenguaje SQL podrá sernos útil en otros entornos y con otros motores de bases de datos, como SQL server de Microsoft, o Access. De modo que lo que aprendamos nos servirá también fuera del ámbito de este curso.

## Instalar el servidor MySQL

Veremos ahora cómo instalar las aplicaciones y paquetes necesarios para poder trabajar con **MySQL**.

Lo primero es obtener el paquete de instalación desde el servidor en Internet: <http://www.mysql.com/>, y después instalarlo en nuestro ordenador.

Se puede descargar el servidor **MySQL** y los clientes estándar directamente desde este enlace: <http://www.mysql.com/downloads/index.html>. Hay que elegir la versión que se quiere descargar, preferiblemente la recomendada, ya que suele ser la más actualizada y estable.

Una vez seleccionada la versión, hay que elegir la distribución adecuada al sistema operativo que usemos: Windows, Linux, Solaris... El fichero descargado se podrá instalar directamente.

Terminar de instalar el servidor depende en gran medida de cada caso, y es mejor remitirse a la documentación facilitada por **MySQL** junto a cada paquete.

### [Instalación según el sistema operativo.](#)

Ahora estamos en disposición de poder utilizar **MySQL** desde la consola de nuestro ordenador, en modo de línea de comandos.

## Y... ¿por qué MySQL?

Ya hemos visto que para acceder a bases de datos es mucho más útil usar un motor o servidor que hace las funciones de intérprete entre las aplicaciones y usuarios con las bases de datos.

Esta utilidad se traduce en ventajas, entre las que podemos mencionar las siguientes:

- Acceso a las bases de datos de forma simultánea por varios usuarios y/o aplicaciones.
- Seguridad, en forma de permisos y privilegios, determinados usuarios tendrán permiso para consulta o modificación de determinadas tablas. Esto permite compartir datos sin que peligre la integridad de la base de datos o protegiendo determinados contenidos.
- Potencia: SQL es un lenguaje muy potente para consulta de bases de datos, usar un motor nos ahorra una enorme cantidad de trabajo.
- Portabilidad: SQL es también un lenguaje estandarizado, de modo que las consultas hechas usando SQL son fácilmente portables a otros sistemas y plataformas. Esto, unido al uso de C/C++ proporciona una portabilidad enorme.

En concreto, usar **MySQL** tiene ventajas adicionales:

- Escalabilidad: es posible manipular bases de datos enormes, del orden de seis mil tablas y alrededor de cincuenta millones de registros, y hasta 32 índices por tabla.
- **MySQL** está escrito en C y C++ y probado con multitud de compiladores y dispone de APIs para muchas plataformas diferentes.
- Conectividad: es decir, permite conexiones entre diferentes máquinas con distintos sistemas operativos. Es corriente que servidores Linux o Unix, usando **MySQL**, sirvan datos para ordenadores con Windows, Linux, Solaris, etc. Para ello se usa TCP/IP, tuberías, o sockets Unix.
- Es multihilo, con lo que puede beneficiarse de sistemas multiprocesador.
- Permite manejar multitud de tipos para columnas.
- Permite manejar registros de longitud fija o variable.

# 1 Definiciones

Pero, ¿qué son las bases de datos?

La teoría es muy compleja y bastante árida, si estás interesado en estudiar una buena base teórica deberías consultar la sección de bibliografía y enlaces. El presente curso sólo tiene por objetivo explicar unas bases generales, aunque sólidas, suficientes para desarrollar la mayor parte de los pequeños y medianos proyectos.

Hay que señalar que existen varios modelos lógicos de bases de datos, aunque en este curso sólo veremos el modelo de **bases de datos relacionales**. (Al menos de momento).

Bien, probablemente tengas una idea intuitiva de lo que es una base de datos, y probablemente te suenen algunos conceptos, como tupla, tabla, relación, clave... Es importante que veamos algunas definiciones, ya que gran parte de la teoría que veremos en este curso se basa en conceptos que tienen significados muy precisos dentro de la teoría de bases de datos.

## Dato

No es sencillo definir qué es un dato, pero intentaremos ver qué es desde el punto de vista de las bases de datos.

Podemos decir que un dato es una información que refleja el valor de una característica de un objeto real, sea concreto o abstracto, o imaginario (nada nos impide hacer una base de datos sobre duendes :-).

Debe cumplir algunas condiciones, por ejemplo, debe permanecer en el tiempo. En ese sentido, estrictamente hablando, una edad no es un dato, ya que varía con el tiempo. El *dato* sería la fecha de nacimiento, y la edad se calcula a partir de ese dato y de la fecha actual. Además, debe tener un significado, y debe ser manipulable mediante operadores: comparaciones, sumas, restas, etc (por supuesto, no todos los datos admiten todos los operadores).

## Base de datos

Podemos considerar que es un conjunto de datos de varios tipos, organizados e interrelacionados. Estos datos deben estar libres de redundancias innecesarias y ser independientes de los programas que los usan.

## SGBD (DBMS)

Son las siglas que significan *Sistema de Gestión de Bases de Datos*, en inglés DBMS, *DataBase Manager System*. En este caso, **MySQL** es un SGBD, o mejor dicho: nuestro SGBD.

## Consulta

Es una petición al SGBD para que procese un determinado comando SQL. Esto incluye tanto peticiones de datos como creación de bases de datos, tablas, modificaciones, inserciones, etc.

## Redundancia de datos

Decimos que hay redundancia de datos cuando la misma información es almacenada varias veces en la misma base de datos. Esto es siempre algo a evitar, la redundancia dificulta la tarea de modificación de datos, y es el motivo más frecuente de inconsistencia de datos. Además requiere un mayor espacio de almacenamiento, que influye en mayor coste y mayor tiempo de acceso a los datos.

## Inconsistencia de datos

Sólo se produce cuando existe redundancia de datos. La inconsistencia consiste en que no todas las copias redundantes contienen la misma información. Así, si existen diferentes modos de obtener la misma información, y esas formas pueden conducir a datos almacenados en distintos sitios. El problema surge al modificar esa información, si lo sólo cambiamos esos valores en algunos de los lugares en que se guardan, las consultas que hagamos más tarde podrán dar como resultado respuestas inconsistentes (es decir, diferentes). Puede darse el caso de que dos aplicaciones diferentes proporcionen resultados distintos para el mismo dato.

## Integridad de datos

Cuando se trabaja con bases de datos, generalmente los datos se reparten entre varios ficheros. Si, como pasa con MySQL, la base de datos está disponible para varios usuarios de forma simultánea, deben existir mecanismos que aseguren que las interrelaciones entre registros se mantienen coherentes, que se respetan las dependencias de existencia y que las claves únicas no se repitan.

Por ejemplo, un usuario no debe poder borrar una entidad de una base de datos, si otro usuario está usando los datos de esa entidad. Este tipo de situaciones son potencialmente peligrosas, ya que provocan situaciones con frecuencia imprevistas. Ciertos errores de integridad pueden provocar que una base de datos deje de ser usable.

Los problemas de integridad se suelen producir cuando varios usuarios están editando datos de la misma base de datos de forma simultánea. Por ejemplo, un usuario crea un nuevo registro, mientras otro edita uno de los existentes, y un tercero borra otro. El DBMS debe asegurar que se pueden

realizar estas tareas sin que se produzcan errores que afecten a la integridad de la base de datos.

## **2 Diseño de bases de datos: El Modelo conceptual El modelo Entidad-Relación**

En este capítulo, y en los siguientes, explicaremos algo de teoría sobre el diseño bases de datos para que sean seguras, fiables y para que tengan un buen rendimiento.

La teoría siempre es algo tedioso, aunque intentaremos que resulte amena, ya que es necesaria. Aconsejo leer con atención estos capítulos. En ellos aprenderemos técnicas como el "modelado" y la "normalización" que nos ayudarán a diseñar bases de datos, mediante la aplicación de ciertas reglas muy sencillas. Aprenderemos a identificar claves y a elegir claves principales, a establecer interrelaciones, a seleccionar los tipos de datos adecuados y a crear índices.

Ilustraremos estos capítulos usando ejemplos sencillos para cada caso, en próximos capítulos desarrollaremos el diseño de algunas bases de datos completas.

Como siempre que emprendamos un nuevo proyecto, grande o pequeño, antes de lanzarnos a escribir código, crear tablas o bases de datos, hay que analizar el problema sobre el papel. En el caso de las bases de datos, pensaremos sobre qué tipo de información necesitamos guardar, o lo que es más importante: qué tipo de información necesitaremos obtener de la base de datos. En esto consiste el modelado de bases de datos.

### **Modelado de bases de datos**

El proceso de trasladar un problema del mundo real a un ordenador, usando bases de datos, se denomina *modelado*.

Para el modelado de bases de datos es necesario seguir un procedimiento determinado. Pero, cuando el problema a modelar es sencillo, con frecuencia estaremos tentados de pasar por alto algunos de los pasos, y crear directamente bases de datos y tablas. En el caso de las bases de datos, como en cualquier otra solución informática, esto es un gran error. Siempre será mejor seguir todos los pasos del diseño, esto nos ahorrará (con toda seguridad) mucho tiempo más adelante. Sobre todo si alguna vez tenemos que modificar la base de datos para corregir errores o para implementar alguna característica nueva, algo que sucede con mucha frecuencia.

Además, seguir todo el proceso nos facilitará una documentación necesaria para revisar o mantener la aplicación, ya sea por nosotros mismos o por otros administradores o programadores.

La primera fase del diseño de una aplicación (la base de datos, generalmente, es parte de una aplicación), consiste en hablar con el cliente para saber qué quiere, y qué necesita realmente.

Esto es una tarea ardua y difícil. Generalmente, los clientes no saben demasiado sobre programación y sobre bases de datos, de modo que normalmente, no saben qué pueden pedir. De hecho, lo más habitual es que ni siquiera sepan qué es lo que necesitan.

Los modelos conceptuales ayudan en esta fase del proyecto, ya que facilitan una forma clara de ver el proceso en su totalidad, puesto que se trata de una representación gráfica. Además, los modelos conceptuales no están orientados a ningún sistema físico concreto: tipo de ordenador, sistema operativo, SGBD, etc. Ni siquiera tienen una orientación informática clara, podrían servir igualmente para explicar a un operario cómo funciona el proceso de forma manual. Esto facilita que sean comprensibles para personas sin conocimientos de programación.

Además de consultar con el cliente, una buena técnica consiste en observar el funcionamiento del proceso que se quiere informatizar o modelar. Generalmente esos procesos ya se realizan, bien de una forma manual, con ayuda de libros o ficheros; o bien con un pequeño apoyo ofimático.

Con las bases de datos lo más importante es observar qué tipo de información se necesita, y que parte de ella se necesita con mayor frecuencia. Por supuesto, modelar ciertos procesos puede proporcionarnos ayudas extra sobre el proceso manual, pero no debemos intentar que nuestra aplicación lo haga absolutamente todo, sino principalmente, aquello que es realmente necesario.

Cuando los programas se crean sin un cliente concreto, ya sea porque se pretende crear un producto para uso masivo o porque sólo lo vamos a usar nosotros, el papel del cliente lo jugaremos nosotros mismos, pero la experiencia nos enseñará que esto no siempre es una ventaja. Es algo parecido a los que pasa con los abogados o los médicos. Se suele decir que "*el abogado que se defiende a si mismo tiene un necio por cliente*". En el caso de los programadores esto no es tan exagerado; pero lo cierto es que, demasiadas veces, los programadores somos nuestros peores clientes.

Toda esta información recogida del cliente debe formar parte de la documentación. Nuestra experiencia como programadores debe servir, además, para ayudar y guiar al cliente. De este modo podemos hacerle ver posibles "cuellos de botella", excepciones, mejoras en el proceso, etc. Así mismo, hay que explicar al cliente qué es exactamente lo que va a obtener. Cuando un cliente recibe un producto que no esperaba, generalmente no se siente muy inclinado a pagar por él.

Una vez recogidos los datos, el siguiente paso es crear un modelo conceptual. El modelo más usado en bases de datos es el *modelo Entidad-Relación*, que es el que vamos a explicar en este capítulo.

Muy probablemente, esta es la parte más difícil de la resolución del problema. Es la parte más "intelectual" del proceso, en el sentido de que es la que más requerirá pensar. Durante esta fase, seguramente, deberemos tomar ciertas decisiones, que en cierto modo limitarán en parte el modelo. Cuando esto suceda, no estará de más consultar con el cliente para que estas decisiones sean, al menos, aceptadas por él, y si es posible, que sea el propio cliente el que las plantee. ;-)

La siguiente fase es convertir el modelo conceptual en un modelo lógico. Existen varios modelos lógicos, pero el más usado, el que mejor se adapta a **MySQL** y el que por lo tanto explicaremos aquí, es el *modelo Relacional*. La conversión entre el modelo conceptual y el lógico es algo bastante mecánico, aunque no por ello será siempre sencillo.

En el caso del modelo lógico relacional, existe un proceso que sirve para verificar que hemos aplicado bien el modelo, y en caso contrario, corregirlo para que sea así. Este proceso se llama *normalización*, y también es bastante mecánico.

El último paso consiste en codificar el modelo lógico en un modelo físico. Este proceso está ligado al DBMS elegido, y es, seguramente, la parte más sencilla de aplicar, aunque nos llevará mucho más tiempo y espacio explicarla, ya que en el caso del DBMS que nos ocupa (**MySQL**), se requiere el conocimiento del lenguaje de consulta SQL.

## Modelo Entidad-Relación

En esencia, el modelo entidad-relación (en adelante E-R), consiste en buscar las entidades que describan los objetos que intervienen en el problema y las relaciones entre esas entidades.

Todo esto se plasma en un esquema gráfico que tiene por objeto, por una parte, ayudar al programador durante la codificación y por otra, al usuario a comprender el problema y el funcionamiento del programa.

## Definiciones

Pero lo primero es lo primero, y antes de continuar, necesitamos entendernos. De modo que definiremos algunos conceptos que se usan en el modelo E-R. Estas definiciones nos serán útiles tanto para explicar la teoría, como para entendernos entre nosotros y para comprender otros textos sobre el modelado de bases de datos. Se trata de conceptos usados en libros y artículos sobre bases de datos, de modo que será interesante conocerlos con precisión.

### Entidad

Estamos hablando del modelo Entidad-Relación, por lo tanto este es un concepto que no podemos dejar sin definir.

**Entidad:** es una representación de un objeto individual concreto del mundo real.

Si hablamos de personas, tu y yo somos entidades, como individuos. Si hablamos de vehículos, se tratará de ejemplares concretos de vehículos, identificables por su matrícula, el número de chasis o el de bastidor.

**Conjunto de entidades:** es la clase o tipo al que pertenecen entidades con características comunes.

Cada individuo puede pertenecer a diferentes conjuntos: habitantes de un país, empleados de una empresa, miembros de una lista de correo, etc. Con

los vehículos pasa algo similar, pueden pertenecer a conjuntos como un parque móvil, vehículos de empresa, etc.

En el modelado de bases de datos trabajaremos con conjuntos de entidades, y no con entidades individuales. La idea es generalizar de modo que el modelo se ajuste a las diferentes situaciones por las que pasará el proceso modelado a lo largo de su vida. Será el usuario final de la base de datos el que trabaje con entidades. Esas entidades constituirán los datos que manejará con la ayuda de la base de datos.

**Atributo:** cada una de las características que posee una entidad, y que agrupadas permiten distinguirla de otras entidades del mismo conjunto.

En el caso de las personas, los atributos pueden ser características como el nombre y los apellidos, la fecha y lugar de nacimiento, residencia, número de identificación... Si se trata de una plantilla de empleados nos interesarán otros atributos, como la categoría profesional, la antigüedad, etc.

En el caso de vehículos, los atributos serán la fecha de fabricación, modelo, tipo de motor, matrícula, color, etc.

Según el conjunto de entidades al que hallamos asignado cada entidad, algunos de sus atributos podrán ser irrelevantes, y por lo tanto, no aparecerán; pero también pueden ser necesarios otros. Es decir, el conjunto de atributos que usaremos para una misma entidad dependerá del conjunto de entidades al que pertenezca, y por lo tanto del proceso modelado.

Por ejemplo, no elegiremos los mismos atributos para personas cuando formen parte de modelos diferentes. En un conjunto de entidades para los socios de una biblioteca, se necesitan ciertos atributos. Estos serán diferentes para las mismas personas, cuando se trate de un conjunto de entidades para los clientes de un banco.

**Dominio:** conjunto de valores posibles para un atributo.

Una fecha de nacimiento o de matriculación tendrá casi siempre un dominio, aunque generalmente se usará el de las fechas posibles. Por ejemplo, ninguna persona puede haber nacido en una fecha posterior a la actual. Si esa persona es un empleado de una empresa, su fecha de nacimiento estará en un dominio tal que actualmente tenga entre 16 y 65 años. (Por supuesto, hay excepciones...)

Los números de matrícula también tienen un dominio, así como los colores de chapa o los fabricantes de automóviles (sólo existe un número limitado de empresas que los fabrican).

Generalmente, los dominios nos sirven para limitar el tamaño de los atributos. Supongamos que una empresa puede tener un máximo de 1000 empleados. Si uno de los atributos es el número de empleado, podríamos decir que el dominio de ese atributo es (0,1000).

Con nombres o textos, los dominios limitarán su longitud máxima.

Sin embargo, los dominios no son demasiado importantes en el modelo E-R, nos preocuparemos mucho más de ellos en el modelo relacional y en el físico.

## Relación

El otro concepto que no podemos dejar de definir es el de relación. Aunque en realidad, salvo para nombrar el modelo, usaremos el término interrelación, ya que *relación* tiene un significado radicalmente diferente dentro del modelo relacional, y esto nos puede llevar a error.

**Interrelación:** es la asociación o conexión entre conjuntos de entidades.

Tengamos los dos conjuntos: de personas y de vehículos. Podemos encontrar una interrelación entre ambos conjuntos a la que llamaremos *posee*, y que asocie una entidad de cada conjunto, de modo que un individuo *posea* un vehículo.

**Grado:** número de conjuntos de entidades que intervienen en una interrelación.

De este modo, en la anterior interrelación intervienen dos entidades, por lo que diremos que es de grado 2 o binaria. También existen interrelaciones de grado 3, 4, etc. Pero las más frecuentes son las interrelaciones binarias.

Podemos establecer una interrelación ternaria (de grado tres) entre personas, de modo que dos personas sean padre y madre, respectivamente, de una tercera.

Existen además tres tipos distintos de interrelaciones binarias, dependiendo del número de entidades del primer conjunto de entidades y del segundo. Así hablaremos de interrelaciones 1:1 (uno a uno), 1:N (uno a muchos) y N:M (muchos a muchos).

Nuestro ejemplo anterior de "persona *posee* vehículo" es una interrelación de 1:N, ya que cada persona puede no poseer vehículo, poseer uno o poseer más de uno. Pero cada vehículo sólo puede ser propiedad de una persona.

Otras relaciones, como el matrimonio, es de 1:1, o la de amistad, de N:M.

## Clave

Estaremos de acuerdo en que es muy importante poder identificar claramente cada entidad y cada interrelación. Esto es necesario para poder

referirnos a cada elemento de un conjunto de entidades o interrelaciones, ya sea para consultarlo, modificarlo o borrarlo. No deben existir ambigüedades en ese sentido.

En principio, cada entidad se puede distinguir de otra por sus atributos. Aunque un subconjunto de atributos puedan ser iguales en entidades distintas, el conjunto completo de todos los atributos no se puede repetir nunca. Pero a menudo son sólo ciertos subconjuntos de atributos los que son diferentes para todas las entidades.

**Clave:** es un conjunto de atributos que identifican de forma unívoca una entidad.

En nuestro ejemplo de las entidades persona, podemos pensar que de una forma intuitiva sabemos qué atributos distinguen a dos personas distintas. Sabemos que el nombre por sí mismo, desde luego, no es uno de esos atributos, ya que hay muchas personas con el mismo nombre. A menudo, el conjunto de nombre y apellidos puede ser suficiente, pero todos sabemos que existen ciertos nombres y apellidos comunes que también se repiten, y que esto es más probable si se trata de personas de la misma familia.

Las personas suelen disponer de un documento de identidad que suele contener un número que es distinto para cada persona. Pero habrá aplicaciones en que este valor tampoco será una opción: podemos tener, por ejemplo, personas en nuestra base de datos de distintas nacionalidades, o puede que no tengamos acceso a esa información (una agenda personal no suele contener ese tipo de datos), también hay personas, como los menores de edad, que generalmente no disponen de documento de identidad.

Con otros tipos de entidad pasa lo mismo. En el caso de vehículos no siempre será necesario almacenar el número de matrícula o de bastidor, o tal vez no sea un valor adecuado para usar como clave (ya veremos más adelante que en el esquema físico es mucho mejor usar valores enteros).

En fin, que en ocasiones, por un motivo u otro, creamos un atributo artificial para usarlo sólo como clave. Esto es perfectamente legal en el modelo E-R, y se hace frecuentemente porque resulta cómodo y lógico.

## Claves candidatas

Una característica que debemos buscar siempre en las claves es que contengan el número mínimo de atributos, siempre que mantengan su función. Diremos que una clave es mínima cuando si se elimina cualquiera de los atributos que la componen, deja de ser clave. Si en una entidad existe más de una de estas claves mínimas, cada una de ellas es una *clave candidata*.

**Clave candidata:** es cada una de las claves mínimas existente en un conjunto de entidades.

## Clave principal

Si disponemos de varias claves candidatas no usaremos cualquiera de ellas según la ocasión. Esto sería fuente de errores, de modo que siempre usaremos la misma clave candidata para identificar la entidad.

**Clave principal:** (o primaria), es una clave candidata elegida de forma arbitraria, que usaremos siempre para identificar una entidad.

## Claves de interrelaciones

Para identificar interrelaciones el proceso es similar, aunque más simple. Tengamos en cuenta que para definir una interrelación usaremos las claves primarias de las entidades interrelacionadas. De este modo, el identificador de una interrelación es el conjunto de las claves primarias de cada una de las entidades interrelacionadas.

Por ejemplo, si tenemos dos personas identificadas con dos valores de su clave primaria, *clave1* y *clave2*, y queremos establecer una interrelación "es padre de" entre ellas, usaremos esas dos claves. El identificador de la interrelación será *clave1,clave2*.

## Entidades fuertes y débiles

A menudo la clave de una entidad está ligada a la clave principal de otra, aún sin tratarse de una interrelación. Por ejemplo, supongamos una entidad viaje, que usa la clave de un vehículo y añade otros atributos como origen, destino, fecha, distancia. Decimos que la entidad viaje es una entidad débil, en contraposición a la entidad vehículo, que es una entidad fuerte. La diferencia es que las entidades débiles no necesitan una clave primaria, sus claves siempre están formadas como la combinación de una clave primaria de una entidad fuerte y otros atributos.

Además, la existencia de las entidades débiles está ligada o subordinada a la de la fuerte. Es decir, existe una dependencia de existencia. Si eliminamos un vehículo, deberemos eliminar también todos los viajes que ese vehículo ha realizado.

## Dependencia de existencia

**Dependencia de existencia:** decimos que existe una dependencia de existencia entre una entidad, *subordinada*, y otra, *dominante*, cuando la eliminación de la entidad dominante, conlleva también la eliminación de la entidad o entidades subordinadas.

Desde cierto punto de vista, podemos considerar que las entidades dominantes y sus entidades subordinadas forman parte de una misma entidad. Es decir, una entidad está formada por ella misma y sus circunstancias (citando a Ortega :-). Esas circunstancias podrían ser, en el caso de nuestro

vehículo, además de los viajes que ha hecho, los dueños que ha tenido, las revisiones que se le han efectuado, averías, etc. Es decir, todo su historial.

## Generalización

**Generalización:** es el proceso según el cual se crea un conjunto de entidades a partir de otros que comparten ciertos atributos.

A veces existen situaciones en que sea conveniente crear una entidad como una fusión de otras, en principio, diferentes, aunque con atributos comunes. Esto disminuye el número de conjuntos de entidades y facilita el establecimiento de interrelaciones.

Por ejemplo, estamos modelando la gestión de una biblioteca, en la que además de libros se pueden consultar y prestar revistas y películas. Desde el punto de vista del modelo E-R, deberíamos crear conjuntos de entidades distintos para estos tres tipos de entidad, sin embargo, todos ellos tienen comportamientos y características comunes: préstamos, ubicaciones, ejemplares, editorial. También tienen atributos específicos, como el número de revista, o la duración de la película.

La idea es crear una entidad con una única copia de los atributos comunes y añadir los atributos no comunes. Además se debe añadir un atributo que indique que tipo de entidad estamos usando, este atributo es un *discriminador*.

La desventaja de la generalización es que se desperdicia espacio de almacenamiento, ya que sólo algunos de los atributos no comunes contienen información en cada entidad, el resto se desperdicia.

La ventaja es que podemos establecer el mismo tipo de interrelación con cualquier entidad del conjunto. En nuestro ejemplo, en lugar de tener que establecer tres interrelaciones de préstamo, o ubicación, bastará con una de cada tipo.

## Especialización

Es el proceso inverso al de generalización, en lugar de crear una entidad a partir de varias, descomponemos una entidad en varias más especializadas.

**Especialización:** es el proceso según el cual se crean varios tipos de entidades a partir de uno. Cada una de los conjuntos de entidades resultantes contendrá sólo algunos de los atributos del conjunto original.

La idea es lógica: si la generalización tiene ventajas e inconvenientes, cuando los inconvenientes superan a las ventajas, será conveniente hacer una especialización.

Por ejemplo, para gestionar la flota de vehículos de una empresa usamos un único conjunto de entidades, de modo que tratamos del mismo modo motocicletas, utilitarios, limusinas, furgonetas y camiones. Pero, desde el punto de vista de mantenimiento, se pueden considerar entidades diferentes: cada una de ellas tiene revisiones distintas y en talleres diferentes. Es decir, las diferencias superan a los atributos comunes. Este conjunto de entidades es un buen candidato a la especialización.

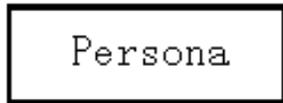
En realidad, es irrelevante si una entidad es fruto de una generalización o de una especialización, no deja de ser una entidad, y por lo tanto, no afecta al modelo.

## Representación de entidades y relaciones: Diagramas

No hay unanimidad total con respecto a la representación de diagramas E-R, he encontrado algunas discrepancias en los distintos documentos que he consultado, dependiendo de la variedad concreta del modelo que usen. Pero a grandes rasgos todos están de acuerdo en lo que expondremos aquí.

### Entidad

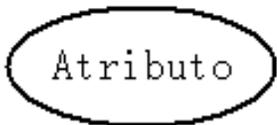
Las entidades se representan con un rectángulo, y en su interior el nombre de la entidad:



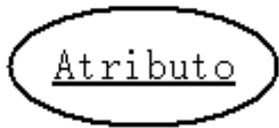
Las entidades débiles pueden representarse mediante dos rectángulos inscritos. Ya sabemos que existe una dependencia de existencia entre la entidad débil y la fuerte, esto se representa también añadiendo una flecha a la línea que llega a la entidad débil.

### Atributo

Los atributos se representan mediante elipses, y en su interior el nombre del atributo:



Algunas variantes de diagramas E-R usan algunas marcas para indicar que cierto atributo es una clave primaria, como subrayar el nombre del atributo.



También es frecuente usar una doble elipse para indicar atributos multivaluados:



**Atributo multivaluado:** (o multivalorado) se dice del atributo tal que para una misma entidad puede tomar varios valores diferentes, es decir, varios valores del mismo dominio.

## Interrelación

Las interrelaciones se representan mediante rombos, y en su interior el nombre de la interrelación:

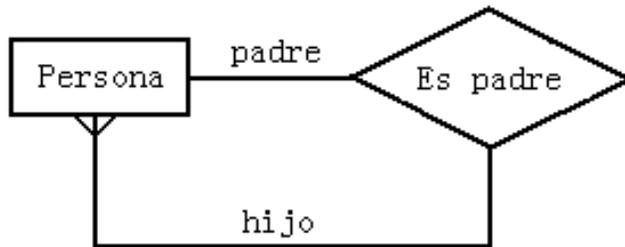


En los extremos de las líneas que parten del rombo se añaden unos números que indican la cantidad de entidades que intervienen en la interrelación: 1, n. Esto también se suele hacer modificando el extremo de las líneas. Si terminan con un extremo involucran a una entidad, si terminan en varios extremos, (generalmente tres), involucrarán a varias entidades:



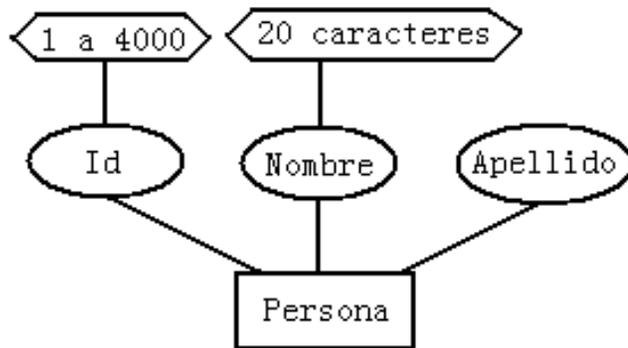


Sobre las líneas a veces se añade el rol que representa cada entidad:



## Dominio

A veces es conveniente añadir información sobre el dominio de un atributo, los dominios se representan mediante hexágonos, con la descripción del dominio en su interior:



## Diagrama

Un diagrama E-R consiste en representar mediante estas figuras un modelo completo del problema, proceso o realidad a describir, de forma que se definan tanto las entidades que lo componen, como las interrelaciones que existen entre ellas.

La idea es simple, aparentemente, pero a la hora de construir modelos sobre realidades concretas es cuando surgen los problemas. La realidad es siempre compleja. Las entidades tienen muchos atributos diferentes, de los cuales debemos aprender a elegir sólo los que necesitamos. Lo mismo cabe decir de las interrelaciones. Además, no siempre está perfectamente claro qué es un atributo y qué una entidad; o que ventajas obtenemos si tratamos a ciertos atributos como entidades y viceversa.

Al final, nuestra mejor arma es la práctica. Cuantos más problemas diferentes modelemos más aprenderemos sobre el proceso y sobre los problemas que pueden surgir. Podremos aplicar la experiencia obtenida en otros proyectos, y, si no reducir el tiempo empleado en el modelado, al menos sí reducir los retoques posteriores, el mantenimiento y el tiempo necesario para realizar modificaciones sobre el modelo.

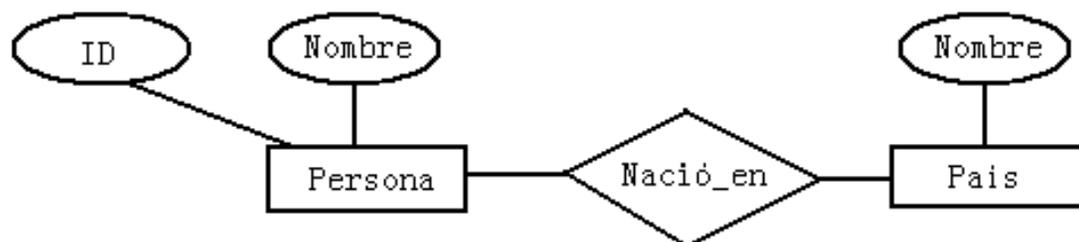
## Construir un modelo E-R

Podemos dividir el proceso de construir un modelo E-R en varias tareas más simples. El proceso completo es iterativo, es decir, una vez terminado debemos volver al comienzo, repasar el modelo obtenido y, probablemente, modificarlo. Una vez satisfechos con el resultado (tanto nosotros, los programadores, como el cliente), será el momento de pasar a la siguiente fase: el modelo lógico.

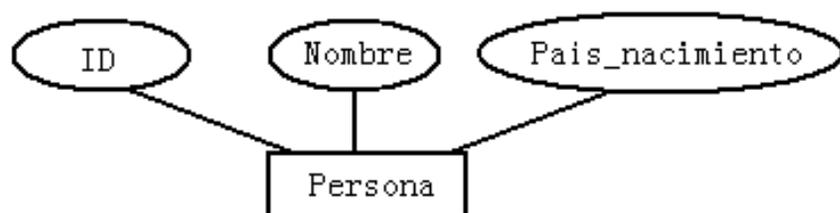
Uno de los primeros problemas con que nos encontraremos será decidir qué son entidades y qué atributos.

La regla principal es que una entidad sólo debe contener información sobre un único objeto real.

Pero en ciertos casos esto nos puede obligar a crear entidades con un único atributo. Por ejemplo, si creamos una entidad para representar una persona, uno de los atributos puede ser el lugar de nacimiento. El lugar de nacimiento: población, provincia o país, puede ser considerado como una entidad. *Bueno, yo creo que un país tiene muchas y buenas razones para ser considerado una entidad.* Sin embargo en nuestro caso concreto, tal vez, esta información sea sólo eso: un lugar de nacimiento. ¿Debemos pues almacenar esa información como un atributo de persona o debemos, por el contrario, crear una entidad independiente?.

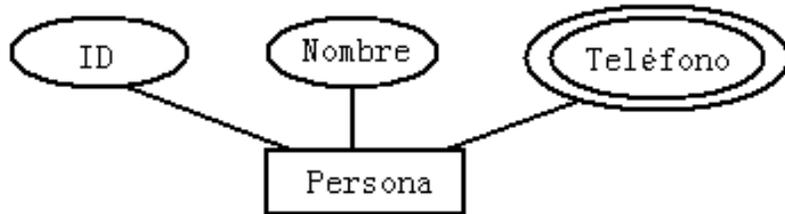


Una regla que puede ayudarnos en esta decisión es que si una entidad sólo tiene un atributo, que sirve para identificarlo, entonces esa entidad puede ser considerada como un atributo.

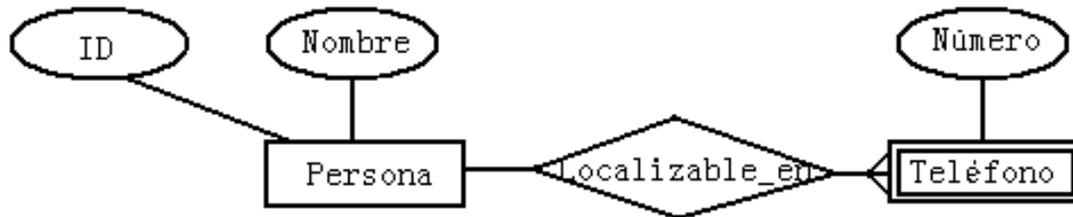


Otro problema frecuente se presenta con los atributos multivaluados.

Por ejemplo, cada persona puede ser localizada en varios números de teléfono. Considerando el teléfono de contacto como un atributo de persona, podemos afirmar que tal atributo es multivaluado.



Pero, aunque como su propio nombre indica no dejan de ser atributos, es mejor considerar a los atributos multivaluados como entidades débiles subordinadas. Esto nos evitará muchos problemas con el modelo lógico relacional.



## Proceso

Para crear un diagrama conceptual hay que meditar mucho. No hay un procedimiento claro y universal, aunque sí se pueden dar algunas directrices generales:

- Hablar con el cliente e intentar dejar claros los parámetros y objetivos del problema o proceso a modelar. Por supuesto, tomar buena nota de todo.
- Estudiar el planteamiento del problema para:
  - Identificar los conjuntos de entidades útiles para modelar el problema,
  - Identificar los conjuntos de interrelaciones y determinar su grado y tipo (1:1, 1:n o m:n).
- Trazar un primer diagrama E-R.
- Identificar atributos y dominios para los conjuntos de entidades e interrelaciones.
- Seleccionar las claves principales para los conjuntos de entidades.
- Verificar que el modelo resultante cumple el planteamiento del problema. Si no es así, se vuelve a repasar el proceso desde principio.
- Seguir con los siguientes pasos: traducir el diagrama a un modelo lógico, etc.

Quiero hacer notar que, la mayor parte del proceso que hemos explicado para crear un diagrama E-R, también puede servir para crear aplicaciones, aunque no estén basadas en bases de datos.

## Extensiones

Existen varias extensiones al modelo E-R que hemos visto, aunque la mayor parte de ellas no las vamos a mencionar.

Una de ellas es la cardinalidad de asignación, que se aplica a atributos multivaluados. Consiste en establecer un número mínimo y máximo de posibles valores para atributos multivaluados.

Por ejemplo, en nuestra entidad persona habíamos usado un atributo multivaluado para los teléfonos de contacto. Habíamos dicho que, para evitar problemas en el modelo lógico, era mejor considerar este atributo como una entidad. Sin embargo hay otras soluciones. Si por ejemplo, establecemos una cardinalidad para este atributo (0,3), estaremos imponiendo, por diseño, que para cada persona sólo puede haber una cantidad entre 0 y 3 de teléfonos de contacto. Si esto es así, podemos usar tres atributos, uno por cada posible teléfono, y de ese modo eliminamos el atributo multivaluado.

No siempre será posible establecer una cardinalidad. En el ejemplo planteado se la eligió de una forma completamente arbitraria, y probablemente no sea una buena idea. En otros casos sí existirá una cardinalidad clara, por ejemplo, en un automóvil con cinco plazas, las personas que viajen en él tendrán una cardinalidad (1,5), al menos tiene que haber un conductor, y como máximo otros cuatro pasajeros.

Otra posible extensión consiste en algo llamado "entidad compuesta". En realidad se trata de una interrelación como las que hemos visto, pero a la que se añaden más atributos.

Por ejemplo, en la relación de matrimonio entre dos personas, podríamos añadir un atributo para guardar la fecha de matrimonio.

## Ejemplo 1

Nos enfrentamos al siguiente problema que debemos modelar.

Se trata de una base de datos que debe almacenar la información sobre varias estaciones meteorológicas, en una zona determinada. De cada una de ellas recibiremos y almacenaremos un conjunto de datos cada día: temperatura máxima y mínima, precipitaciones en litros/m<sup>2</sup>, velocidad del viento máxima y mínima, y humedad máxima y mínima.

El sistema debe ser capaz de seleccionar, añadir o eliminar estaciones. Para cada una almacenaremos su situación geográfica (latitud y longitud), identificador y altitud.

Bien, es un problema sencillo, pero nos sirve para ilustrar el procedimiento.

Ya tenemos la descripción del proceso, así que pasemos al segundo paso:

## Identificar conjuntos de entidades

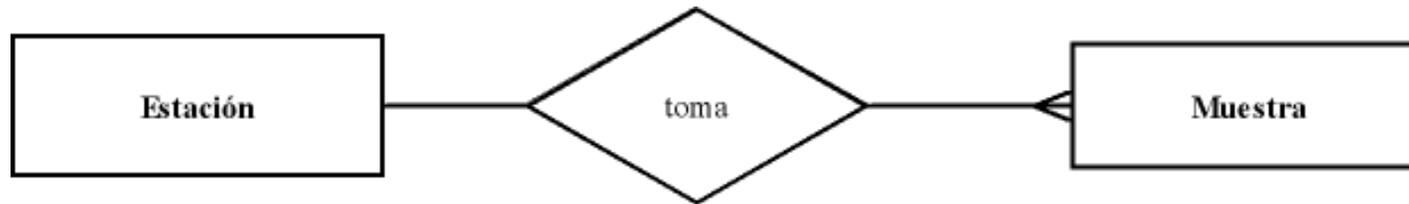
A primera vista, tenemos dos conjuntos de entidades: estaciones y muestras. Podríamos haber usado sólo un conjunto, el de las muestras, pero nos dicen que debemos ser capaces de seleccionar, añadir y borrar estaciones, de modo que parece que tendremos que usar un conjunto de entidades para ellas.

## Identificar conjuntos de interrelaciones

Las relaciones son más simples, ya que sólo hay una: cada estación estará interrelacionada con varias muestras. Es una relación 1:N.

## Trazar primer diagrama

Podemos trazar ya, por lo tanto, nuestro primer diagrama:



## Identificar atributos

El siguiente paso es identificar los atributos para cada conjunto de entidades.

Para las muestras tendremos que elegir los que nos da el enunciado: temperatura máxima y mínima, precipitaciones, velocidades del viento máxima y mínima y humedad máxima y mínima. Además hay que añadir la fecha de la muestra.

Para las estaciones también nos dicen qué atributos necesitamos: identificador, latitud, longitud y altitud.

## Seleccionar claves principales

Ahora toca seleccionar claves principales.

Las estaciones disponen de varias claves candidatas. Tenemos, por una parte, el identificador, que es único para cada estación, y por otra su situación geográfica, ya que no puede haber dos estaciones en el mismo sitio. Parece lógico usar la primera como clave principal, ya que es un único atributo.

Pero en el caso de las muestras no existen claves candidatas claras. De hecho, el conjunto total de atributos puede no ser único: dos estaciones próximas geográficamente, podrían dar los mismos datos para las mismas fechas.

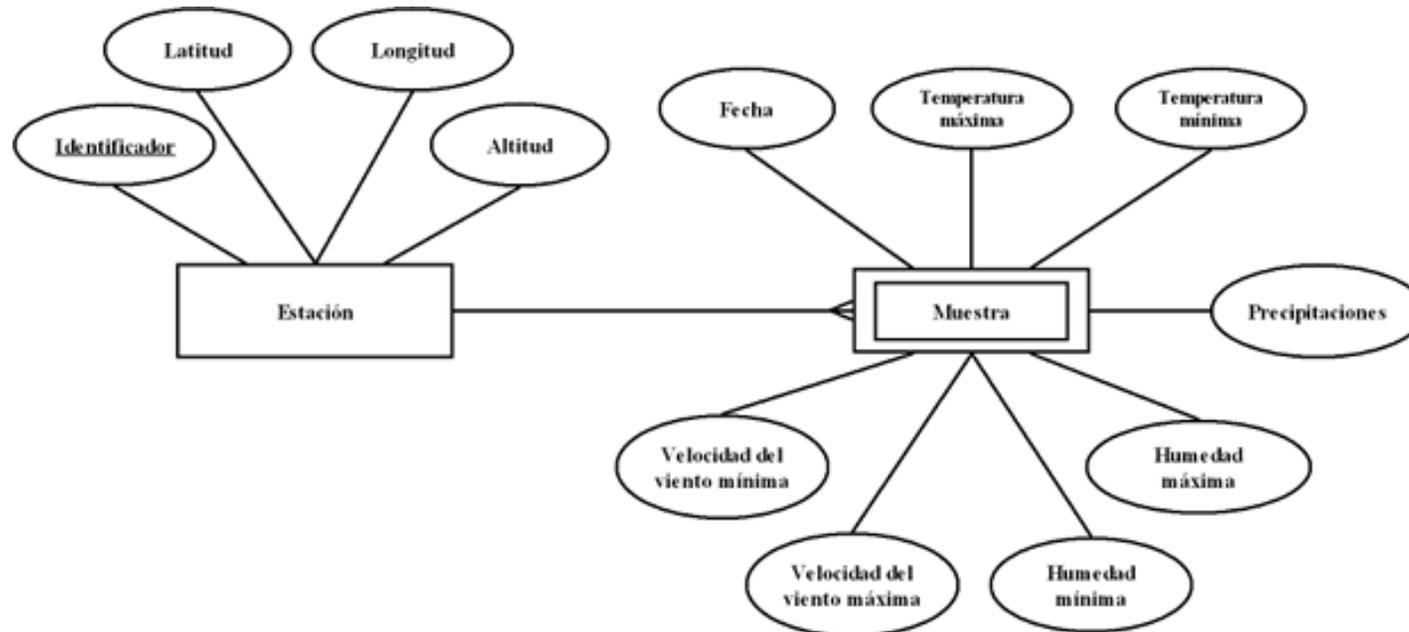
Tenemos una opción para solucionar el problema: crear una clave principal artificial, un número entero que se incremente de forma automática para cada muestra.

Sin embargo esta no es la solución óptima. Otra alternativa es considerar las muestras como entidades débiles subordinadas a las entidades estación. En ese caso, la clave primaria de la estación se almacena como una clave foránea en cada muestra.

Como entidad débil, las muestras no necesitan una clave primaria, de hecho, esa clave se forma con la unión de la clave primaria de la estación y la fecha de la muestra.

La primera solución es factible, pero precisa el uso de un atributo artificial, y como vemos, no absolutamente necesario. Optaremos por la segunda solución.

### Verificar el modelo



Todo está conforme el enunciado, nuestro diagrama E-R está terminado.

### Ejemplo 2

Nuestro segundo ejemplo es más complicado. Se trata de gestionar una biblioteca, y nuestro cliente quiere tener ciertas herramientas a su disposición para controlar libros, socios y préstamos. Adicionalmente se necesita un control de los ejemplares de cada libro, su ubicación y su estado, con vistas a su retirada o restitución, para esto último necesita información sobre editoriales a las que se deben pedir los libros.

Tanto los libros como los socios estarán sujetos a ciertas categorías, de modo que según ellas cada libro podrá ser o no prestado a cada socio. Por ejemplo, si las categorías de los libros van de A a F, y la de los socios de B a F, un libro de categoría A nunca puede ser prestado a ningún socio. Estos libros sólo se pueden consultar en la biblioteca, pero no pueden salir de ella. Un libro de categoría B sólo a socios de categoría B, un libro de categoría C se podrá prestar a socios de categorías B y C, etc. Los libros de categoría F siempre pueden prestarse.

El sistema debe proporcionar también un método de búsqueda para libros por parte de los socios, por tema, autor o título. El socio sólo recibirá información sobre los libros de los que existen ejemplares, y sobre la categoría.

Además, se debe conservar un archivo histórico de préstamos, con las fechas de préstamo y devolución, así como una nota que el responsable de la biblioteca quiera hacer constar, por ejemplo, sobre el estado del ejemplar después de su devolución. Este archivo es una herramienta para la biblioteca que se puede usar para discriminar a socios "poco cuidadosos".

Los préstamos, generalmente, terminan con la devolución del libro, pero algunas veces el ejemplar se pierde o el plazo supera un periodo de tiempo establecido y se da por perdido. Estas circunstancias pueden cerrar un préstamo y provocan la baja del ejemplar (y en ocasiones la del socio :-). Nuestro archivo histórico debe contener información sobre si el libro fue devuelto o perdido.

## Identificar conjuntos de entidades

En lo que se refiere a los libros, podemos considerar los siguientes conjuntos:

1. **Libro:** contiene la información relativa a las publicaciones.
2. **Tema:** temas en los que clasificaremos los contenidos de los libros, esto facilitará las búsquedas y consultas por parte de los socios.

Sobre el tema habría mucho que hablar. Podríamos guardar el tema como un atributo de libro, pero en la práctica será deseable considerar este atributo como una entidad, y establecer una interrelación entre el libro y los temas. Esto nos permitiría elegir varios temas para cada libro, y nos facilitará las búsquedas de libros.

3. **Autor:** contiene información sobre autores, hay que tener en cuenta que un libro puede no tener autor (caso de libros anónimos), y también puede tener uno o varios autores (recopilaciones y colaboraciones).
4. **Editorial:** necesitamos un conjunto de entidades para ellas, ya que deberemos hacer pedidos de libros y esos pedidos se hacen a editoriales.
5. **Ejemplar:** para cada libro podemos tener uno o varios ejemplares. Cada ejemplar es una entidad, de hecho, es posible que tengamos entidades en nuestra base de datos de libros de los que no tenemos ningún ejemplar. Esto también nos permitirá hacer seguimiento de ejemplares concretos, mantener información sobre distintas ediciones del mismo libro, el estado en que se encuentra, diferentes formatos del libro (papel, informático o sonoro), etc.

Otros conjuntos de entidades serán:

6. **Socio:** con la información sobre cada socio.

Podemos tener ciertas dudas sobre si ciertas características del modelo son entidades o relaciones. Por ejemplo, el préstamo puede ser considerado como una interrelación entre un socio y un ejemplar. Sin embargo, necesitaremos alguna información extra sobre cada préstamo, como la fecha de préstamo y la de devolución.

De hecho, lo consideraremos como una entidad compuesta, que relacionará un socio con un ejemplar, (no prestamos libros, sino ejemplares) y al que añadiremos un atributo con la fecha de préstamo.

7. **Préstamo:** entidad compuesta con información sobre cada préstamo.

Otro concepto difícil de catalogar puede ser el de las categorías. Puede ser un atributo o una entidad, dependiendo de si necesitamos más atributos para definirlos o no. Hay que tener en cuenta que tanto los libros como los socios pertenecen a una categoría, lo que tenemos que decidir es si se trata del mismo conjunto de categorías o no.

En principio, y si no nos vemos obligados a cambiar el modelo, parece que lo más lógico es considerar las categorías como un atributo de las entidades libro y socio.

Nuestro cliente, además, dice que quiere conservar un archivo histórico de los préstamos. Necesitamos por lo tanto, otro conjunto de entidades para el histórico de préstamos. Sin embargo, al buscar las interrelaciones veremos que no necesitamos esta octava entidad.

## Identificar conjuntos de interrelaciones

Llega el turno de buscar las interrelaciones entre los conjuntos de entidades que hemos definido. Las primeras que encontramos son las existentes entre *libro* y *tema*, *libro* y *autor*, del tipo N:M.

Entre *libro* y *editorial*, existe una relación N:1.

Entre *libro* y *ejemplar* la relación es del tipo 1:N. En este caso, además, los ejemplares son entidades subordinadas a libro. La interrelación, por lo tanto se convierte en una entidad compuesta.

Podríamos haber establecido un conjunto de relaciones entre *ejemplar* y *editorial*, del tipo 1:N; en lugar de hacerlo entre *libro* y *editorial*. Esto es si consideramos posible que un mismo libro se edite por varias editoriales, pero para este modelo consideraremos que un mismo libro editado por distintas editoriales, son en realidad varios libros diferentes.

Otro conjunto de interrelaciones es la que existe entre *ejemplar* y *socio*, de tipo N:M. Ya dijimos que esta interrelación será en realidad una entidad compuesta.

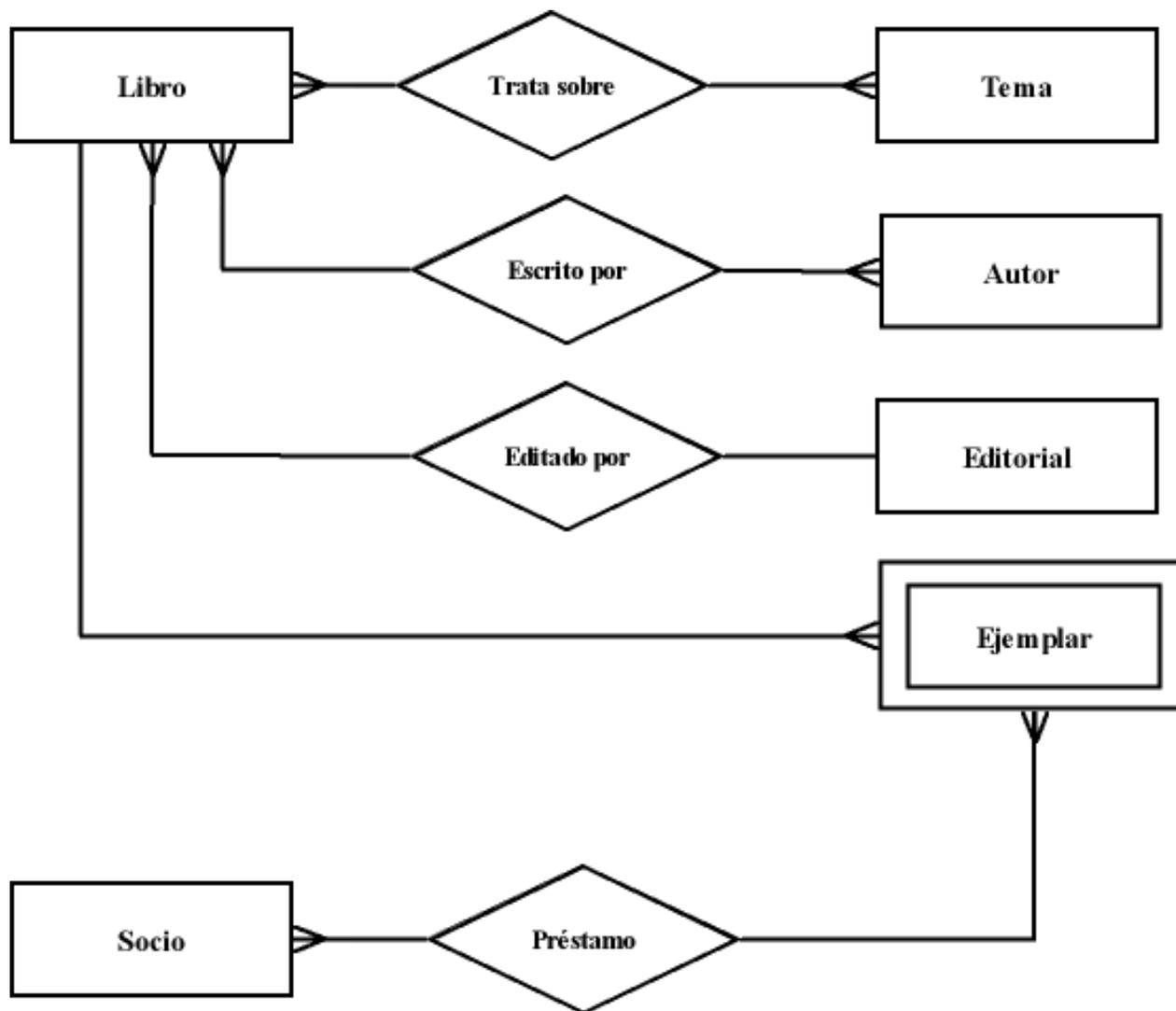
El último conjunto de interrelaciones es entre *préstamo* y su *historial*, en este caso, consideraremos el historial como una entidad subordinada a *préstamo*.

Pero consideremos la posibilidad de hacer una generalización entre las entidades *préstamo* e *historial*, a fin de cuentas, una entidad de *historial* es un *préstamo* cerrado, contendrá la misma información que un préstamo abierto, más algunos datos sobre el cierre.

De este modo tampoco necesitaremos un atributo discriminador creado al efecto, podemos considerar que si la fecha de devolución es nula, el préstamo está abierto.

## Trazar primer diagrama

Trazaremos ahora nuestro primer diagrama:



## Identificar atributos

Iremos entidad por entidad:

1. **Libro:** guardaremos el título. Además, como el título no es único, ya existen muchos libros con el mismo título, necesitaremos un atributo para usar como clave primaria, la clavelibro. Añadiremos también algunos atributos como el idioma, título e idioma originales (para traducciones), formato, etc. Podemos añadir algún otro atributo informativo, como el código ISBN, que facilita su petición a la editorial.
2. **Tema:** igual que con la entidad *libro*, guardaremos el nombre del tema y un atributo clavetema para usar como clave primaria.
3. **Autor:** el nombre del autor. Podemos añadir algún dato sobre el autor, biográfico o de cierta utilidad para realizar búsquedas en detalle. También crearemos un atributo para usarlo como clave principal, al que llamaremos claveautor.
4. **Editorial:** atributos interesantes para hacer peticiones de nuevos libros: nombre, dirección y teléfono.

5. **Ejemplar:** edición, ubicación en la biblioteca. También necesitaremos un identificador de orden, para poder distinguir entre varios ejemplares idénticos.
6. **Socio:** guardaremos el nombre, fecha de alta, categoría y, por supuesto, un identificador para usar como clave primaria: el número de socio.
7. **Préstamo/historial:** necesitamos la fecha de préstamo y de devolución, así como un atributo de comentarios.

## Identificar claves principales

No todas las entidades necesitarán una clave principal, veamos cuales necesitamos:

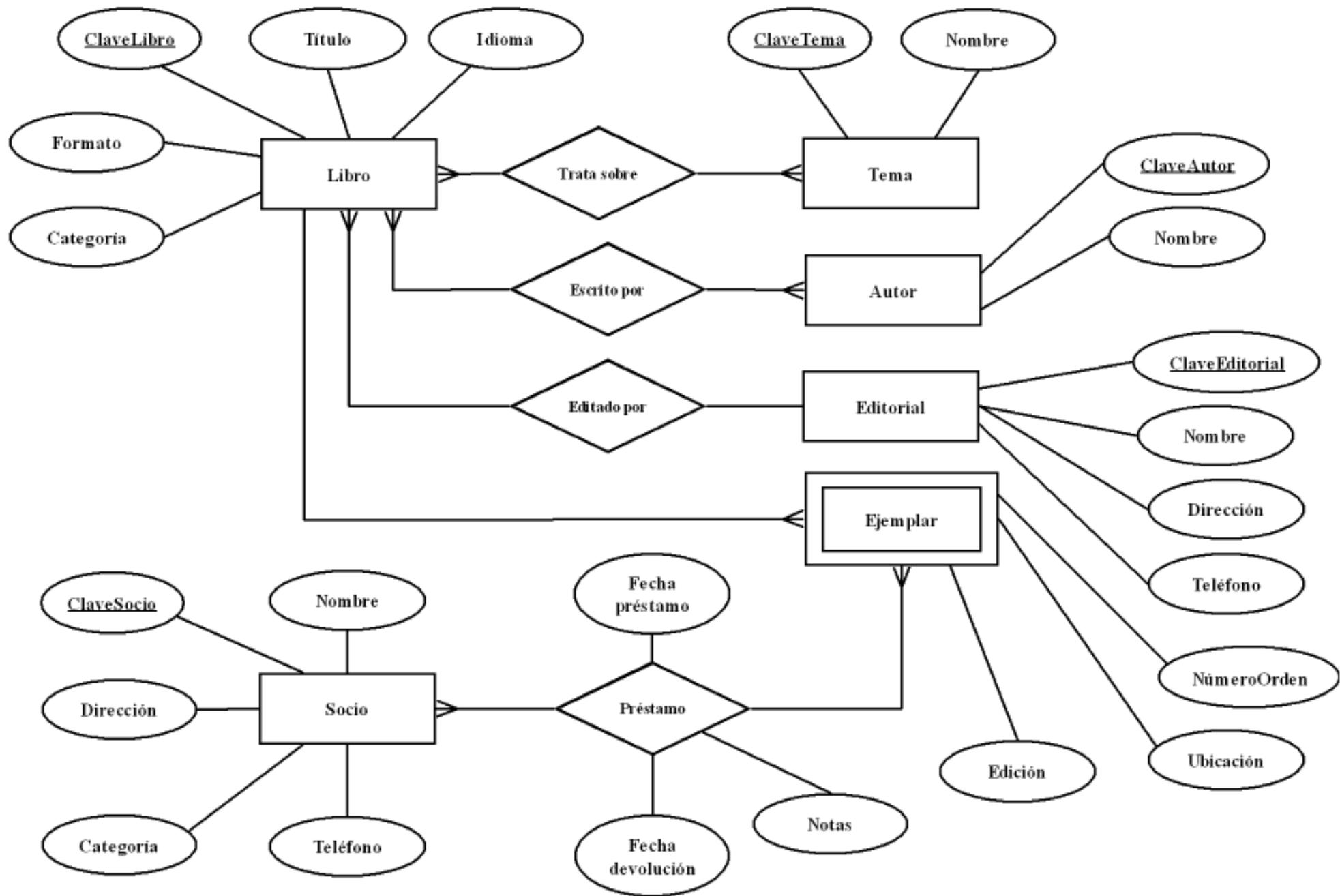
En el caso de **libro**, **tema**, **autor**, **editorial** y **socio** disponemos de atributos creados específicamente para ser usados como claves principales.

Para **ejemplar**, en principio, al tratarse de una entidad subordinada, no necesitaría una clave principal, pero como interviene en la interrelación de **préstamo** sí la necesitaremos. Esta clave se crea mediante la combinación de la clave principal de **libro** y un atributo propio de **ejemplar**, que será el identificador de orden.

La entidad **préstamo/historial** es compuesta. En principio, con las claves de socio y ejemplar debería ser suficiente para identificar un préstamo concreto, pero es posible que el mismo socio pida prestado el mismo ejemplar más de una vez. Estaremos, por lo tanto, obligados a usar un atributo extra para definir una clave candidata. Ese atributo puede ser, y de hecho debe ser, la fecha de préstamo. De todos modos, esta entidad no precisa de clave principal, ya que no interviene en ninguna interrelación. Si fuese necesaria, no sería posible usar el conjunto de atributos de clave de socio, clave de libro, número de orden y fecha, ya que algunos de ellos podrían ser nulos (si se dan de baja socios o ejemplares).

## Verificar el modelo

Trazaremos ahora nuestro diagrama completo:



## 3 Diseño de bases de datos:

### El modelo lógico

### El modelo relacional

#### Modelo relacional

Entre los modelos lógicos, el modelo relacional está considerado como el más simple. No vamos a entrar en ese tema, puesto que es el único que vamos a ver, no tiene sentido establecer comparaciones.

Diremos que es el que más nos conviene. Por una parte, el paso del modelo E-R al relacional es muy simple, y por otra, **MySQL**, como implementación de SQL, está orientado principalmente a bases de datos relacionales.

El doctor Edgar F. Codd, un investigador de IBM, inventó en 1970 el *modelo relacional*, también desarrolló el sistema de *normalización*, que veremos en el siguiente capítulo.

El modelo se compone de tres partes:

1. Estructura de datos: básicamente se compone de relaciones.
2. Manipulación de datos: un conjunto de operadores para recuperar, derivar o modificar los datos almacenados.
3. Integridad de datos: una colección de reglas que definen la consistencia de la base de datos.

#### Definiciones

Igual que hicimos con el modelo E-R, empezaremos con algunas definiciones. Algunos de los conceptos son comunes entre los dos modelos, como atributo o dominio. Pero de todos modos, los definiremos de nuevo.

#### Relación

Es el concepto básico del modelo relacional. Ya adelantamos en el capítulo anterior que los conceptos de relación entre el modelo E-R y el relacional son diferentes. Por lo tanto, usamos el término *interrelación* para referirnos a la conexión entre entidades. En el modelo relacional este término se refiere a una tabla, y es el paralelo al concepto conjunto de entidades del modelo E-R.

**Relación:** es un conjunto de datos referentes a un conjunto de entidades y organizados en forma tabular, que se compone de filas y columnas,

(tuplas y atributos), en la que cada intersección de fila y columna contiene un valor.

## Tupla

A menudo se le llama también *registro* o *fila*, físicamente es cada una de las líneas de la relación. Equivale al concepto de entidad del modelo E-R, y define un objeto real, ya sea abstracto, concretos o imaginario.

**Tupla:** cada una de las filas de una relación. Contiene la información relativa a una única entidad.

De esta definición se deduce que no pueden existir dos tuplas iguales en la misma relación.

## Atributo

También denominado *campo* o *columna*, corresponde con las divisiones verticales de la relación. Corresponde al concepto de atributo del modelo E-R y contiene cada una de las características que definen una entidad u objeto.

**Atributo:** cada una de las características que posee una entidad, y que agrupadas permiten distinguirla de otras entidades del mismo conjunto.

Al igual que en el modelo E-R, cada atributo tiene asignado un nombre y un dominio. El conjunto de todos los atributos es lo que define a una entidad completa, y es lo que compone una tupla.

## Nulo (*NULL*)

Hay ciertos atributos, para determinadas entidades, que carecen de valor. El modelo relacional distingue entre valores vacíos y valores nulos. Un valor vacío se considera un valor tanto como cualquiera no vacío, sin embargo, un nulo *NULL* indica la ausencia de valor.

**Nulo:** (*NULL*) valor asignado a un atributo que indica que no contiene ninguno de los valores del dominio de dicho atributo.

El nulo es muy importante en el modelo relacional, ya que nos permite trabajar con datos desconocidos o ausentes.

Por ejemplo, si tenemos una relación de vehículos en la que podemos guardar tanto motocicletas como automóviles, un atributo que indique a qué lado está el volante (para distinguir vehículos con el volante a la izquierda de los que lo tienen a la derecha), carece de sentido en motocicletas. En ese caso, ese atributo para entidades de tipo motocicleta será *NULL*.

Esto es muy interesante, ya que el dominio de este atributo es (derecha,izquierda), de modo que si queremos asignar un valor del dominio no hay otra opción. El valor nulo nos dice que ese atributo no tiene ninguno de los valores posibles del dominio. Así que, en cierto modo amplía la información.

Otro ejemplo, en una relación de personas tenemos un atributo para la fecha de nacimiento. Todas las personas de la relación han nacido, pero en un determinado momento puede ser necesario insertar una para la que desconocemos ese dato. Cualquier valor del dominio será, en principio, incorrecto. Pero tampoco será posible distinguirlo de los valores correctos, ya que será una fecha. Podemos usar el valor *NULL* para indicar que la fecha de nacimiento es desconocida.

## Dominio

**Dominio:** Rango o conjunto de posibles valores de un atributo.

El concepto de dominio es el mismo en el modelo E-R y en el modelo relacional. Pero en este modelo tiene mayor importancia, ya que será un dato importante a la hora de dimensionar la relación.

De nuevo estamos ante un concepto muy flexible. Por ejemplo, si definimos un atributo del tipo entero, el dominio más amplio sería, lógicamente, el de los números enteros. Pero este dominio es infinito, y sabemos que los ordenadores no pueden manejar infinitos números enteros. Al definir un atributo de una relación dispondremos de distintas opciones para guardar datos enteros. Si en nuestro caso usamos la variante de "entero pequeño", el dominio estará entre -128 y 127. Pero además, el atributo corresponderá a una característica concreta de una entidad; si se tratase, por ejemplo, de una calificación sobre 100, el dominio estaría restringido a los valores entre 0 y 100.

## Modelo relacional

Ahora ya disponemos de los conceptos básicos para definir en qué consiste el modelo relacional. Es un modelo basado en relaciones, en la que cada una de ellas cumple determinadas condiciones mínimas de diseño:

- No deben existir dos tuplas iguales.
- Cada atributo sólo puede tomar un único valor del dominio, es decir, no pueden contener listas de valores.
- El orden de las tuplas dentro de la relación y el de los atributos, dentro de cada tupla, no es importante.

## Cardinalidad

**Cardinalidad:** número de tuplas que contiene una relación.

La cardinalidad puede cambiar, y de hecho lo hace frecuentemente, a lo largo del tiempo: siempre se pueden añadir y eliminar tuplas.

## Grado

**Grado:** número de atributos de cada tupla.

El grado de una relación es un valor constante. Esto no quiere decir que no se puedan agregar o eliminar atributos de una relación; lo que significa es que si se hace, la relación cambia. Cambiar el grado, generalmente, implicará modificaciones en las aplicaciones que hagan uso de la base de datos, ya que cambiarán conceptos como claves e interrelaciones, de hecho, puede cambiar toda la estructura de la base de datos.

## Esquema

**Esquema:** es la parte *constante* de una relación, es decir, su estructura.

Esto es, el esquema es una lista de los atributos que definen una relación y sus dominios.

## Instancia

**Instancia:** es el conjunto de las tuplas que contiene una relación en un momento determinado.

Es como una fotografía de la relación, que sólo es válida durante un periodo de tiempo concreto.

## Clave

**Clave:** es un conjunto de atributos que identifica de forma unívoca a una tupla. Puede estar compuesto por un único atributo o una combinación de varios.

Dentro del modelo relacional no existe el concepto de clave múltiple. Cada clave sólo puede hacer referencia a una tupla de una tabla. Por lo tanto, todas las claves de una relación son únicas.

Podemos clasificar las claves en distintos tipos:

- *Candidata:* cada una de las posibles claves de una relación, en toda relación existirá al menos una clave candidata. Esto implica que ninguna relación puede contener tuplas repetidas.
- *Primaria:* (o principal) es la clave candidata elegida por el usuario para identificar las tuplas. No existe la necesidad, desde el punto de vista de la teoría de bases de datos relacionales, de elegir una clave primaria. Además, las claves primarias no pueden tomar valores nulos. Es preferible, por motivos de optimización de **MySQL**, que estos valores sean enteros, aunque no es obligatorio. **MySQL** sólo admite una clave primaria por tabla, lo cual es lógico, ya que la definición implica que sólo puede existir una.
- *Alternativa:* cada una de las claves candidatas que no son clave primaria, si es que existen.
- *Foránea:* (o externa) es el atributo (o conjunto de atributos) dentro de una relación que contienen claves primarias de otra relación. No hay nada que impida que ambas relaciones sean la misma.

## Interrelación

Decimos que dos relaciones están interrelacionadas cuando una posee una clave foránea de la otra. Cada una de las claves foráneas de una relación establece una interrelación con la relación donde esa clave es la principal.

Según esto, existen dos tipos de interrelación:

- La interrelación entre entidades fuertes y débiles.
- La interacción pura, entre entidades fuertes.

Extrictamente hablando, sólo la segunda es una interrelación, pero como veremos más tarde, en el modelo relacional ambas tienen la forma de relaciones, al igual que las entidades compuestas, que son interrelaciones con atributos añadidos.

Al igual que en el modelo E-R, existen varios tipos de interrelación:

- *Uno a uno:* a cada tupla de una relación le corresponde una y sólo una tupla de otra.
- *Uno a varios:* a cada tupla una relación le corresponden varias en otra.
- *Varios a varios:* cuando varias tuplas de una relación se pueden corresponder con varias tuplas en otra.

## Paso del modelo E-R al modelo relacional

Existen varias reglas para convertir cada uno de los elementos de los diagramas E-R en tablas:

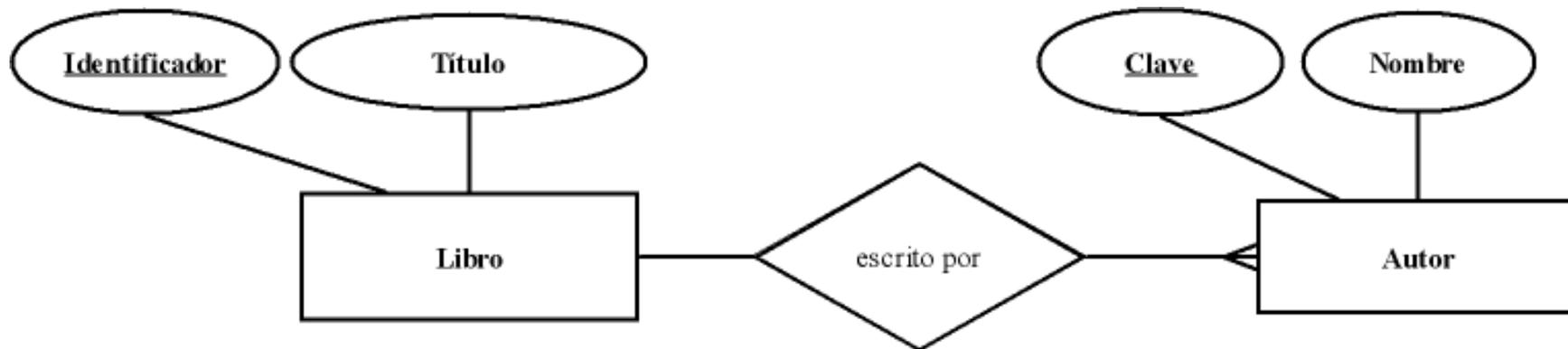
1. Para cada conjunto de entidades fuertes se crea una relación con una columna para cada atributo.
2. Para cada conjunto de entidades débiles se crea una relación que contiene una columna para los atributos que forman la clave primaria de la entidad fuerte a la que se encuentra subordinada y una columna para cada atributo de la entidad.
3. Para cada interrelación se crea una relación que contiene una columna para cada atributo correspondiente a las claves principales de las entidades interrelacionadas.
4. Lo mismo para entidades compuestas, añadiendo las columnas necesarias para los atributos añadidos a la interrelación.

Las relaciones se representan mediante sus esquemas, la sintaxis es simple:

```
<nombre_relación>(<nombre_atributo_i>,...)
```

La clave principal se suele indicar mediante un subrayado.

Veamos un ejemplo, partamos del siguiente diagrama E-R:



Siguiendo las normas indicadas obtendremos las siguientes relaciones:

```
Libro(Identificador, Título)
Autor(Clave, Nombre)
Escrito_por(Identificador, Clave)
```

## Manipulación de datos, álgebra relacional

El modelo relacional también define el modo en que se pueden manipular las relaciones. Hay que tener en cuenta que este modelo tiene una base matemática muy fuerte. Esto no debe asustarnos, en principio, ya que es lo que le proporciona su potencia y seguridad. Es cierto que también complica su estudio, pero afortunadamente, no tendremos que comprender a fondo la teoría para poder manejar el modelo.

En el modelo relacional define ciertos operadores. Estos operadores relacionales trabajan con tablas, del mismo modo que los operadores matemáticos trabajan con números. Esto implica que el resultado de las operaciones con relaciones son relaciones, lo cual significa que, como veremos, que no necesitaremos implementar bucles.

El álgebra relacional define el modo en que se aplican los operadores relacionales sobre las relaciones y los resultados que se obtienen. Del mismo modo que al aplicar operadores enteros sobre números enteros sólo da como salida números enteros, en álgebra relacional los resultados de aplicar operadores son relaciones.

Disponemos de varios operadores, que vamos a ver a continuación.

### Selección

Se trata de un operador unitario, es decir, se aplica a una relación y como resultado se obtiene otra relación.

Consiste en seleccionar ciertas tuplas de una relación. Generalmente la selección se limita a las tuplas que cumplan determinadas condiciones.

```
<relación>[<atributo>='<valor>']
```

Por ejemplo, tengamos la siguiente relación:

```
tabla(id, nombre, apellido, fecha, estado)
```

<b>tabla</b>	<u>id</u>	<u>nombre</u>	<u>apellido</u>	<u>fecha</u>	<u>estado</u>
	123	Fulano	Prierez	4/12/1987	soltero
	454	Mengano	Sianchiez	15/1/1990	soltero
	102	Tulana	Liopez	24/6/1985	casado
	554	Filgana	Gaomez	15/5/1998	soltero

```
005 Tutulano Gionziales 2/6/1970 viudo
```

Algunos ejemplos de selección serían:

```
tabla[id<'200']
```

<u>id</u>	<u>nombre</u>	<u>apellido</u>	<u>fecha</u>	<u>estado</u>
123	Fulano	Prierez	4/12/1987	soltero
102	Tulana	Liopez	24/6/1985	casado
005	Tutulano	Gionziales	2/6/1970	viudo

```
tabla[estado='soltero']
```

123	Fulano	Prierez	4/12/1987	soltero
454	Mengano	Sianchiez	15/1/1990	soltero
554	Filgana	Gaomez	15/5/1998	soltero

## Proyección

También es un operador unitario.

Consiste en seleccionar ciertos atributos de una relación.

Esto puede provocar un conflicto. Como la relación resultante puede no incluir ciertos atributos que forman parte de la clave principal, existe la posibilidad de que haya tuplas duplicadas. En ese caso, tales tuplas se eliminan de la relación de salida.

```
<relación>[<lista de atributos>]
```

Por ejemplo, tengamos la siguiente relación:

```
tabla(id, nombre, apellido, fecha, estado)
```

```
tabla
```

<u>id</u>	<u>nombre</u>	<u>apellido</u>	<u>fecha</u>	<u>estado</u>
-----------	---------------	-----------------	--------------	---------------

123	Fulano	Prierez	4/12/1987	soltero
454	Mengano	Sianchiez	15/1/1990	soltero
102	Tulana	Liopez	24/6/1985	casado
554	Fulano	Gaomez	15/5/1998	soltero
005	Tutulano	Gionziales	2/6/1970	viudo

Algunos ejemplos de proyección serían:

**tabla[id, apellido]**

<u>id</u>	<u>apellido</u>
123	Prierez
454	Sianchiez
102	Liopez
554	Gaomez
005	Gionziales

**tabla[nombre, estado]**

<u>nombre</u>	<u>estado</u>
Fulano	soltero
Mengano	soltero
Tulana	casado
Tutulano	viudo

En esta última proyección se ha eliminado una tupla, ya que aparece repetida. Las tuplas 1ª y 4ª son idénticas, las dos personas de nombre 'Fulano' son solteras.

## Producto cartesiano

Este es un operador binario, se aplica a dos relaciones y el resultado es otra relación.

El resultado es una relación que contendrá todas las combinaciones de las tuplas de los dos operandos.

Esto es: si partimos de dos relaciones, R y S, cuyos grados son n y m, y cuyas cardinalidades a y b, la relación producto tendrá todos los atributos presentes en ambas relaciones, por lo tanto, el grado será n+m. Además la cardinalidad será el producto de a y b.

Para ver un ejemplo usaremos dos tablas inventadas al efecto:

```

tabla1(id, nombre, apellido)
tabla2(id, número)

```

```

tabla1
id nombre apellido
15 Fulginio Liepez
26 Cascanio Suanchiez

```

```

tabla2
id número
15 12345678
26 21222112
15 66525425

```

El resultado del producto cartesiano de tabla1 y tabla2: tabla1 x tabla2 es:

```

tabla1 x tabla2
id nombre apellido id número
15 Fulginio Liepez 15 12345678
26 Cascanio Suanchiez 15 12345678
15 Fulginio Liepez 26 21222112
26 Cascanio Suanchiez 26 21222112
15 Fulginio Liepez 15 66525425
26 Cascanio Suanchiez 15 66525425

```

Podemos ver que el grado resultante es  $3+2=5$ , y la cardinalidad  $2*3 = 6$ .

## Composición (Join)

Una composición (Join en inglés) es una restricción del producto cartesiano, en la relación de salida sólo se incluyen las tuplas que cumplan una determinada condición.

La condición que se usa más frecuentemente es la igualdad entre dos atributos, uno de cada tabla.

```

<relación1>[<condición>]<relación2>

```

Veamos un ejemplo. Partimos de dos relaciones:

```
tabla1(id, nombre, apellido)
tabla2(id, número)
```

```
tabla1
id  nombre  apellido
15  Fulginio Liepez
26  Cascanio Suanchiez
```

```
tabla2
id  número
15  12345678
26  21222112
15  66525425
```

La composición de estas dos tablas, para una condición en que 'id' sea igual en ambas sería:

```
tabla1[tabla1.id = tabla2.id]tabla2
id  nombre  apellido  t2.id  número
15  Fulginio Liepez    15     12345678
26  Cascanio Suanchiez 26     21222112
15  Fulginio Liepez    15     66525425
```

## Composición natural

Cuando la condición es la igualdad entre atributos de cada tabla, la relación de salida tendrá parejas de columnas con valores iguales, por lo tanto, se podrá eliminar siempre una de esas columnas. Cuando se eliminan, el tipo de composición se denomina *composición natural*.

El grado, por lo tanto, en una composición natural es  $n+m-i$ , siendo  $i$  el número de atributos comparados entre ambas relaciones. La cardinalidad de la relación de salida depende de la condición.

Si sólo se compara un atributo, el grado será  $n+m-1$ , si se comparan dos atributos, el grado será  $n+m-2$ , y generalizando, si se comparan  $i$  atributos, el grado será  $n+m-i$ .

En el ejemplo anterior, si hacemos una composición natural, la columna t2.a1 no aparecería, ya que está repetida:

La composición natural de estas dos tablas, para una condición en que 'id' sea igual en ambas sería:

```
tabla1[tabla1.id = tabla2.id]tabla2
id  nombre      apellido      número
15  Fulginio     Liepez        12345678
26  Cascanio     Suanchiez     21222112
15  Fulginio     Liepez        66525425
```

Podemos hacer una composición natural en la que intervengan dos atributos.

```
tabla1(x, y, nombre)
tabla2(x, y, número)
```

```
tabla1
x  y  nombre
A  4  Fulginio
A  6  Cascanio
B  3  Melania
C  4  Juaninia
C  7  Antononio
D  2  Ferninio
D  5  Ananinia
```

```
tabla2
x  y  número
A  3  120
A  6  145
B  2  250
B  5  450
C  4  140
D  2  130
D  5  302
```

Si la condición es que tanto 'x' como 'y' sean iguales en ambas tablas, tendríamos:

```
tabla1[tabla1.x = tabla2.x Y tabla1.y = tabla2.y]tabla2
```

x	y	nombre	número
A	6	Cascanio	145
C	4	Juaninia	140
D	2	Ferninio	130
D	5	Ananinia	302

## Unión

También se trata de un operador binario.

Una unión es una suma. Ya sabemos que para poder sumar, los operandos deben ser del mismo tipo (no podemos sumar peras y limones), es decir, las relaciones a unir deben tener el mismo número de atributos, y además deben ser de dominios compatibles. El grado de la relación resultante es el mismo que el de las relaciones a unir, y la cardinalidad es la suma de las cardinalidades de las relaciones.

```
<relación1> U <relación2>
```

Por ejemplo, tengamos estas tablas:

```
tabla1(id, nombre, apellido)
tabla2(id, nombre, apellido)
```

tabla1		
<u>id</u>	<u>nombre</u>	<u>apellido</u>
15	Fernandio	Garcidia
34	Augustido	Lipoez
12	Julianino	Sianchiez
43	Carlanios	Pierez

tabla2		
<u>id</u>	<u>nombre</u>	<u>apellido</u>
44	Rosinia	Ortiegaz
63	Anania	Pulpez
55	Inesiana	Diominguez

La unión de ambas tablas es posible, ya que tienen el mismo número y tipo de atributos:

#### **tabla1 U tabla2**

<u>id</u>	<u>nombre</u>	<u>apellido</u>
15	Fernandio	Garcidia
34	Augustido	Lipoez
12	Julianino	Sianchiez
43	Carlanios	Pierez
44	Rosinia	Ortiegaz
63	Anania	Pulpez
55	Inesiana	Diominguez

## Intersección

El operador de intersección también es binario.

Para que dos relaciones se puedan interseccionar deben cumplir las mismas condiciones que para que se puedan unir. El resultado es una relación que contendrá sólo las tuplas presentes en ambas relaciones.

```
<relación1> intersección <relación2>
```

Por ejemplo, tengamos estas tablas:

```
tabla1(id, prenda, color)
tabla2(id, prenda, color)
```

<u>id</u>	<u>prenda</u>	<u>color</u>
10	Jersey	Blanco
20	Jersey	Azul
30	Pantalón	Verde
40	Falda	Roja
50	Falda	Naranja

#### **tabla2**

<u>id</u>	<u>prenda</u>	<u>color</u>
15	Jersey	Violeta
20	Jersey	Azul
34	Pantalón	Amarillo
40	Falda	Roja
52	Falda	Verde

Es posible obtener la intersección de ambas relaciones, ya que tienen el mismo número y tipo de atributos:

```
tabla1 intersección tabla2
id   prenda  color
20    Jersey    Azul
40    Falda     Roja
```

## Diferencia

Otro operador binario más.

Los operandos también deben cumplir las mismas condiciones que para la unión o la intersección. El resultado es una relación que contiene las tuplas de la primera relación que no estén presentes en la segunda.

```
<relación1> - <relación2>
```

Por ejemplo, tengamos estas tablas:

```
tabla1(id, prenda, color)
tabla2(id, prenda, color)
```

```
tabla1
id   prenda  color
10    Jersey    Blanco
20    Jersey    Azul
30    Pantalón Verde
40    Falda     Roja
```

50	Falda	Naranja
----	-------	---------

**tabla2**

<u>id</u>	<u>prenda</u>	<u>color</u>
15	Jersey	Violeta
20	Jersey	Azul
34	Pantalón	Amarillo
40	Falda	Roja
52	Falda	Verde

Es posible obtener la diferencia de ambas relaciones, ya que tienen el mismo número y tipo de atributos:

**tabla1 - tabla2**

<u>id</u>	<u>prenda</u>	<u>color</u>
10	Jersey	Blanco
30	Pantalón	Verde
50	Falda	Naranja

## División

La operación inversa al producto cartesiano.

Este tipo de operación es poco frecuente, las relaciones que intervienen como operandos deben cumplir determinadas condiciones, de divisibilidad, que hace difícil encontrar situaciones en que se aplique.

## Integridad de datos

Es muy importante impedir situaciones que hagan que los datos no sean accesibles, o que existan datos almacenados que no se refieran a objetos o entidades existentes, etc. El modelo relacional también provee mecanismos para mantener la integridad. Podemos dividir estos mecanismos en dos categorías:

- Restricciones estáticas, que se refieren a los estados válidos de datos almacenados.
- Restricciones dinámicas, que definen las acciones a realizar para evitar ciertos efectos secundarios no deseados cuando se realizan operaciones de modificación o borrado de datos.

## Restricciones sobre claves primarias

En cuanto a las restricciones estáticas, las más importantes son las que afectan a las claves primarias.

Ninguna de las partes que componen una clave primaria puede ser *NULL*.

Que parte de una clave primaria sea *NULL* indicaría que, o bien esa parte no es algo absolutamente necesario para definir la entidad, con lo cual no debería formar parte de la clave primaria, o bien no sabemos a qué objeto concreto nos estamos refiriendo, lo que implica que estamos tratando con un grupo de entidades. Esto va en contra de la norma que dice que cada tupla contiene datos sólo de una entidad.

Las modificaciones de claves primarias deben estar muy bien controladas.

Dado que una clave primaria identifica de forma unívoca a una tupla en una relación, parece poco lógico que exista necesidad de modificarla, ya que eso implicaría que no estamos definiendo la misma entidad.

Además, hay que tener en cuenta que las claves primarias se usan frecuentemente para establecer interrelaciones, lo cual implica que sus valores se usan en otras relaciones. Si se modifica un valor de una clave primaria hay que ser muy cuidadoso con el efecto que esto puede tener en todas las relaciones en las que se guarden esos valores.

Existen varias maneras de limitar la modificación de claves primarias. Codd apuntó tres posibilidades:

- Que sólo un número limitado de usuarios puedan modificar los valores de claves primarias. Estos usuarios deben ser conscientes de las repercusiones de tales cambios, y deben actuar de modo que se mantenga la integridad.
- La prohibición absoluta de modificar los valores de claves primarias. Modificarlas sigue siendo posible, pero mediante un mecanismo indirecto. Primero hay que eliminar las tuplas cuyas claves se quieren modificar y a continuación darlas de alta con el nuevo valor de clave primaria.
- La creación de un comando distinto para modificar atributos que son claves primarias o partes de ellas, del que se usa para modificar el resto de los atributos.

Cada SGBD puede implementar alguno o varios de estos métodos.

## Integridad referencial

La integridad referencial se refiere a las claves foráneas. Recordemos que una clave foránea es un atributo de una relación, cuyos valores se corresponden con los de una clave primaria en otra o en la misma relación. Este mecanismo se usa para establecer interrelaciones.

La integridad referencial consiste en que si un atributo o conjunto de atributos se define como una clave foránea, sus valores deben existir en la tabla en que ese atributo es clave principal.

Las situaciones donde puede violarse la integridad referencial es en el borrado de tuplas o en la modificación de claves principales. Si se elimina una tupla cuya clave primaria se usa como clave foránea en otra relación, las tuplas con esos valores de clave foránea contendrán valores sin referenciar.

Existen varias formas de asegurarse de que se conserva la integridad referencial:

- **Restringir operaciones:** borrar o modificar tuplas cuya clave primaria es clave foránea en otras tuplas, sólo estará permitido si no existe ninguna tupla con ese valor de clave en ninguna otra relación.  
Es decir, si el valor de una clave primaria en una tupla es "clave1", sólo podremos eliminar esa tupla si el valor "clave1" no se usa en ninguna otra tupla, de la misma relación o de otra, como valor de clave foránea.
- **Transmisión en cascada:** borrar o modificar tuplas cuya clave primaria es clave foránea en otras implica borrar o modificar las tuplas con los mismos valores de clave foránea.  
Si en el caso anterior, modificamos el valor de clave primaria "clave1" por "clave2", todas las apariciones del valor "clave1" en donde sea clave foránea deben ser sustituidos por "clave2".
- **Poner a nulo:** cuando se elimine una tupla cuyo valor de clave primaria aparece en otras relaciones como clave foránea, se asigna el valor *NULL* a dichas claves foráneas.  
De nuevo, siguiendo el ejemplo anterior, si eliminamos la tupla con el valor de clave primaria "clave1", en todas las tuplas donde aparezca ese valor como clave foránea se sustituirá por *NULL*.

Veremos con mucho más detalle como se implementan estos mecanismos en **MySQL** al estudiar el lenguaje SQL.

## Propagación de claves

Se trata de un concepto que se aplica a interrelaciones N:1 ó 1:1, que nos ahorra la creación de una relación. Supongamos las siguientes relaciones, resultado del paso del ejemplo 2 del modelo E-R al modelo relacional:

```
Libro(ClaveLibro, Título, Idioma, Formato, Categoría)
Editado_por(ClaveLibro, ClaveEditorial)
Editorial(ClaveEditorial, Nombre, Dirección, Teléfono)
```

Cada libro sólo puede estar editado por una editorial, la interrelación es N:1. En este caso podemos prescindir de la relación *Editado\_por* añadiendo un atributo a la relación *Libro*, que sea la clave primaria de la editorial:

```
Libro(ClaveLibro, Título, Idioma, Formato, Categoría, ClaveEditorial)
Editorial(ClaveEditorial, Nombre, Dirección, Teléfono)
```

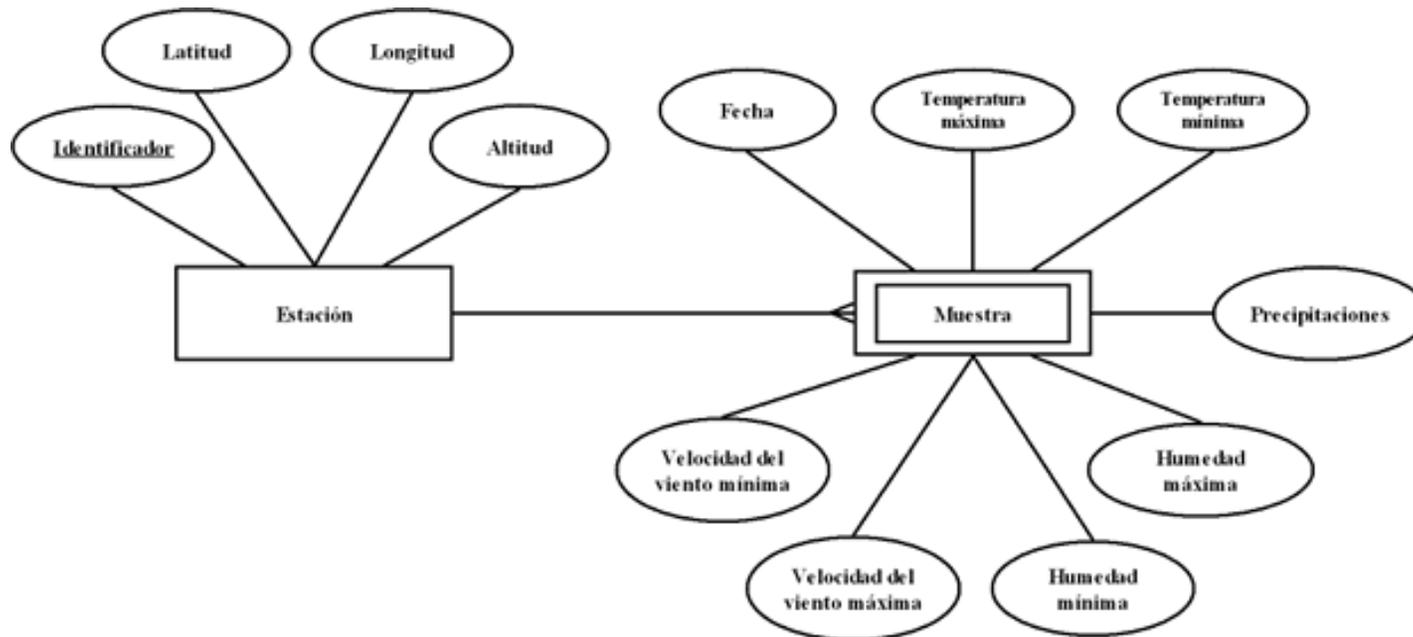
A esto se le denomina propagación de la clave de la entidad *Editorial* a la entidad *Libro*.

Por supuesto, este mecanismo no es válido para interrelaciones de N:M, como por ejemplo, la que existe entre *Libro* y *Autor*.

## Ejemplo 1

Para ilustrar el paso del modelo E-R al modelo relacional, usaremos los mismos ejemplos que en el capítulo anterior, y convertiremos los diagramas E-R a tablas.

Empecemos con el primer ejemplo:



Sólo necesitamos dos tablas para hacer la conversión:

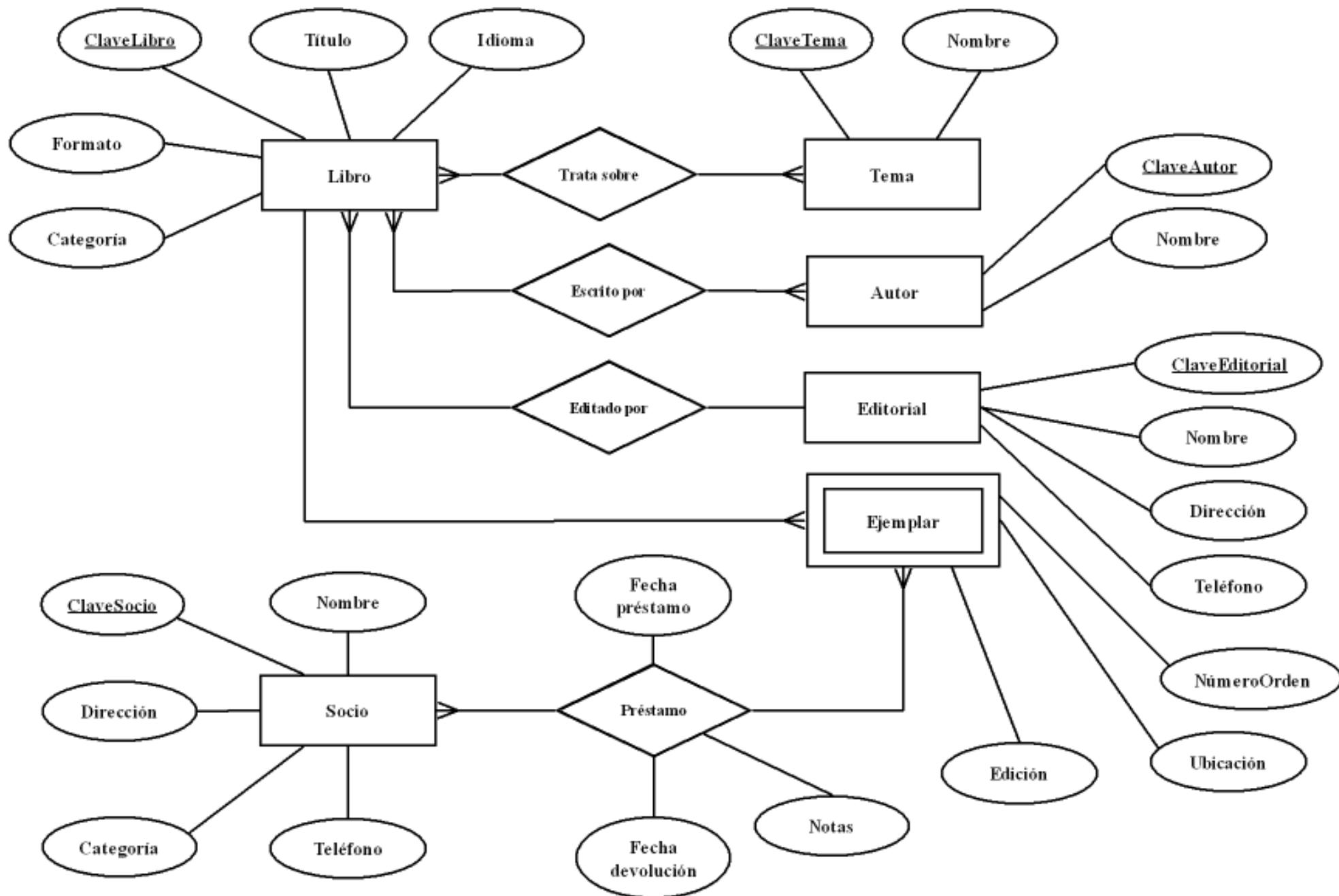
```

Estación(Identificador, Latitud, Longitud, Altitud)
Muestra(IdentificadorEstacion, Fecha, Temperatura mínima, Temperatura máxima,
  Precipitaciones, Humedad mínima, Humedad máxima, Velocidad del viento mínima,
  Velocidad del viento máxima)
  
```

Este ejemplo es muy simple, la conversión es directa. Se puede observar cómo hemos introducido un atributo en la relación *Muestra* que es el

identificador de estación. Este atributo se comporta como una clave foránea.

## Ejemplo 2



Este ejemplo es más complicado, de modo que iremos por fases. Para empezar, convertiremos los conjuntos de entidades en relaciones:

```

Libro(ClaveLibro, Título, Idioma, Formato, Categoría)
Tema(ClaveTema, Nombre)
Autor(ClaveAutor, Nombre)
Editorial(ClaveEditorial, Nombre, Dirección, Teléfono)
Ejemplar(ClaveLibro, NúmeroOrden, Edición, Ubicación)
Socio(ClaveSocio, Nombre, Dirección, Teléfono, Categoría)

```

Recordemos que *Préstamo* es una entidad compuesta:

```

Préstamo(ClaveSocio, ClaveLibro, NúmeroOrden, Fecha_préstamo,
          Fecha_devolución, Notas)

```

La entidad *Ejemplar* es subordinada de *Libro*, es decir, que su clave principal se construye a partir de la clave principal de *Libro* y el atributo NúmeroOrden.

Ahora veamos la conversión de las interrelaciones:

```

Trata_sobre(ClaveLibro, ClaveTema)
Escrito_por(ClaveLibro, ClaveAutor)
Editado_por(ClaveLibro, ClaveEditorial)

```

Ya vimos que podemos aplicar la propagación de claves entre conjuntos de entidades que mantengan una interrelación N:1 ó 1:1. En este caso, la interrelación entre *Libro* y *Editorial* cumple esa condición, de modo que podemos eliminar una interrelación y propagar la clave de *Editorial* a la entidad *Libro*.

El esquema final queda así:

```

Libro(ClaveLibro, Título, Idioma, Formato, Categoría, ClaveEditorial)
Tema(ClaveTema, Nombre)
Autor(ClaveAutor, Nombre)
Editorial(ClaveEditorial, Nombre, Dirección, Teléfono)
Ejemplar(ClaveLibro, NúmeroOrden, Edición, Ubicación)

```

```
Socio(ClaveSocio, Nombre, Dirección, Teléfono, Categoría)
Préstamo(ClaveSocio, ClaveLibro, NúmeroOrden, Fecha_préstamo,
Fecha_devolución, Notas)
Trata_sobre(ClaveLibro, ClaveTema)
Escrito_por(ClaveLibro, ClaveAutor)
```

## 4 Modelo Relacional: Normalización

Llegamos al último proceso netamente teórico del modelado de bases de datos: la normalización. La normalización no es en realidad una parte del diseño, sino más bien una herramienta de verificación. Si hemos diseñado bien los modelos conceptual y lógico de nuestra bases de datos, veremos que la normalización generalmente no requerirá cambios en nuestro diseño.

### Normalización

Antes de poder aplicar el proceso de normalización, debemos asegurarnos de que estamos trabajando con una base de datos relacional, es decir, que cumple con la definición de base de datos relacional.

El proceso de normalización consiste verificar el cumplimiento de ciertas reglas que aseguran la eliminación de redundancias e inconsistencias. Esto se hace mediante la aplicación de ciertos procedimientos y en ocasiones se traduce en la separación de los datos en diferentes relaciones. Las relaciones resultantes deben cumplir ciertas características:

- Se debe conservar la información:
  - Conservación de los atributos.
  - Conservación de las tuplas, evitando la aparición de tuplas que no estaban en las relaciones originales.
- Se deben conservar las dependencias.

Este proceso se lleva a cabo aplicando una serie de reglas llamadas "*formas normales*".

Estas reglas permiten crear bases de datos libres de redundancias e inconsistencias, que se ajusten a la definición del doctor Codd de base de datos relacional.

**MySQL** usa bases de datos relacionales, de modo que deberemos aprender a usar con soltura, al menos, las tres primeras formas normales.

La teoría completa de bases de datos relacionales es muy compleja, y puede resultar muy oscura si se expresa en su forma más simbólica. Ver esta teoría en profundidad está fuera de los objetivos de este curso, al menos por el momento.

Sin embargo, necesitaremos comprender bien parte de esa teoría para aplicar las tres primeras formas normales. La comprensión de las formas cuarta y quinta requieren una comprensión más profunda de conceptos complejos, como el de las *dependencias multivaluadas* o las *dependencias de proyección*.

Generalmente, en nuestras aplicaciones con bases de datos (que podríamos calificar como domésticas

o pequeñas), no será necesario aplicar las formas normales cuarta y quinta, de modo que, aunque las explicaremos lo mejor que podamos, no te preocupes si no consigues aplicarlas por completo.

Por otra parte, si estos conceptos no queden suficientemente claros la culpa es completamente nuestra, por no ser capaces de explicarlos claramente.

Pero ánimo, seguro que en poco tiempo aprendemos a crear bases de datos fuertes y robustas. Pero antes, y sintiéndolo mucho, debemos seguir con algunas definiciones más.

## Primera forma normal (1FN)

**Definición:** Para que una base de datos sea 1FN, es decir, que cumpla la primera forma normal, cada columna debe ser atómica.

Atómica no tiene que ver con la energía nuclear :-), no se trata de crear columnas explosivas ni que produzcan grandes cantidades de energía y residuos radiactivos...

Atómica significa "indivisible", es decir, cada *atributo* debe contener un único valor del *dominio*. Los atributos, en cada tabla de una base de datos 1FN, no pueden tener listas o arrays de valores, ya sean del mismo dominio o de dominios diferentes.

Además, cada atributo debe tener un nombre único. Esto es algo que en general, al menos trabajando con **MySQL**, no nos preocupa; ya que la creación de las tablas implica definir cada columna de un tipo concreto y con un nombre único.

Tampoco pueden existir tuplas idénticas. Esto puede parecer obvio, pero no siempre es así. Supongamos una base de datos para la gestión de la biblioteca, y que el mismo día, y al mismo socio, se le presta dos veces el mismo libro (evidentemente, el libro es devuelto entre cada préstamo, claro). Esto producirá, si no tenemos cuidado, dos registros iguales. Debemos evitar este tipo de situaciones, por ejemplo, añadiendo un atributo con un identificador único de préstamo.

Como vemos, las restricciones de la primera forma normal coinciden con las condiciones de las relaciones de una base de datos relacional, por lo tanto, siempre es obligatorio aplicar esta forma normal.

Aplicar la primera forma normal es muy simple, bastará con dividir cada columna no atómica en tantas columnas atómicas como sea necesario. Por ejemplo, si tenemos una relación que contiene la información de una agenda de amigos con este *esquema*:

```
Agenda(Nombre, email)
```

El nombre, normalmente, estará compuesto por el tratamiento (señor, señora, don, doña, excelencia, alteza, señoría, etc), un nombre de pila y los apellidos. Podríamos considerar el nombre como un dato atómico, pero puede interesarnos separar algunas de las partes que lo componen.

¿Y qué pasa con la dirección de correo electrónico? También podemos considerar que es un valor no atómico, la parte a la izquierda del símbolo @ es el usuario, y a la derecha el dominio. De nuevo, dependiendo de las necesidades del cliente o del uso de los datos, podemos estar interesados en dividir este campo en dos, o incluso en tres partes (puede interesar separar la parte a la derecha del punto en el dominio).

Tanto en esta forma normal, como en las próximas que veremos, es importante no llevar el proceso de normalización demasiado lejos. Se trata de facilitar el trabajo y evitar problemas de redundancia e integridad, y no de lo contrario. Debemos considerar las ventajas o necesidades de aplicar cada norma en cada caso, y no excedernos por intentar aplicar las normas demasiado al pie de la letra.

El esquema de la relación puede quedar como sigue:

```
Agenda(Nombre_Tratamiento, Nombre_Pila, Nombre_Apellidos, email)
```

Otro caso frecuente de relaciones que no cumplen 1FN es cuando existen atributos multivaluados, si todos los valores se agrupan en un único atributo:

```
Libros(Titulo, autores, fecha, editorial)
```

Hemos previsto, muy astutamente, que un libro puede tener varios autores. No es que sea muy frecuente pero sucede, y más con libros técnicos y libros de texto.

Sin embargo, esta relación no es 1FN, ya que en la columna de autores sólo debe existir un valor del dominio, por lo tanto debemos convertir ese atributo en uno multivaluado:

### **Libros**

<u>Titulo</u>	<u>autor</u>	<u>fecha</u>	<u>editorial</u>
Que bueno es MySQL	fulano	12/10/2003	La buena
Que bueno es MySQL	mengano	12/10/2003	La buena
Catástrofes naturales	tulano	18/03/1998	Penútriga

## Dependencias funcionales

Ya hemos comentado que una relación se compone de atributos y dependencias. Los atributos son fáciles de identificar, ya que forman parte de la estructura de la relación, y además, los elegimos nosotros mismos como diseñadores de la base de datos.

Pero no es tan sencillo localizar las dependencias, ya que requieren un análisis de los atributos, o con más precisión, de las interrelaciones entre atributos, y frecuentemente la intuición no es suficiente a la hora de encontrar y clasificar todas las dependencias.

La teoría nos puede ayudar un poco en ese sentido, clasificando las dependencias en distintos tipos, indicando qué características tiene cada tipo.

Para empezar, debemos tener claro que las dependencias se pueden dar entre atributos o entre subconjuntos de atributos.

Estas dependencias son consecuencia de la estructura de la base de datos y de los objetos del mundo real que describen, y no de los valores actualmente almacenados en cada relación. Por ejemplo, si tenemos una relación de vehículos en la que almacenamos, entre otros atributos, la cilindrada y el color, y en un determinado momento todos los vehículos con 2000 c.c. son de color rojo, no podremos afirmar que existen una dependencia entre el color y la cilindrada. Debemos suponer que esto es sólo algo casual.

Para buscar dependencias, pues, no se deben analizar los datos, sino los entes a los que se refieren esos datos.

**Definición:** Sean  $X$  e  $Y$  subconjuntos de atributos de una relación. Diremos que  $Y$  tiene una dependencia funcional de  $X$ , o que  $X$  determina a  $Y$ , si cada valor de  $X$  tiene asociado siempre un único valor de  $Y$ .

El hecho de que  $X$  determine a  $Y$  no quiere decir que conociendo  $X$  se pueda conocer  $Y$ , sino que en la relación indicada, cada vez que el atributo  $X$  tome un determinado valor, el atributo  $Y$  en la misma tupla siempre tendrá el mismo valor.

Por ejemplo, si tenemos una relación con clientes de un hotel, y dos de sus atributos son el número de cliente y su nombre, podemos afirmar que el nombre tiene una dependencia funcional del número de cliente. Es decir, cada vez que en una tupla aparezca determinado valor de número de cliente, es seguro que el nombre de cliente será siempre el mismo.

La dependencia funcional se representa como  $X \rightarrow Y$ .

El símbolo  $\rightarrow$  se lee como "implica" o "determina", y la dependencia anterior se lee como *X implica Y* o *X determina Y*.

Podemos añadir otro símbolo a nuestra álgebra de dependencias: el símbolo  $\nrightarrow$  significa negación. Así  $X \nrightarrow Y$  se lee como *X no determina Y*.

## Dependencia funcional completa

**Definición:** En una dependencia funcional  $X \rightarrow Y$ , cuando  $X$  es un conjunto de atributos, decimos que la dependencia funcional es completa, si sólo depende de  $X$ , y no de ningún subconjunto de  $X$ .

La dependencia funcional se representa como  $X \Rightarrow Y$ .

## Dependencia funcional elemental

**Definición:** Si tenemos una dependencia completa  $X \Rightarrow Y$ , diremos que es una dependencia funcional elemental si  $Y$  es un atributo, y no un conjunto de ellos.

Estas son las dependencias que buscaremos en nuestras relaciones. Las dependencias funcionales elementales son un caso particular de las dependencias completas.

## Dependencia funcional trivial

**Definición:** Una dependencia funcional  $A \rightarrow B$  es trivial cuando  $B$  es parte de  $A$ . Esto sucede cuando  $A$  es un conjunto de atributos, y  $B$  es a su vez un subconjunto de  $A$ .

## Segunda forma normal (2FN)

**Definición:** Para que una base de datos sea 2FN primero debe ser 1FN, y además todas las

columnas que formen parte de una *clave candidata* deben aportar información sobre la clave completa.

Esta regla significa que en una relación sólo se debe almacenar información sobre un tipo de entidad, y se traduce en que los atributos que no aporten información directa sobre la clave principal deben almacenarse en una relación separada.

Lo primero que necesitamos para aplicar esta forma normal es identificar las claves candidatas.

Además, podemos elegir una clave principal, que abreviaremos como **PK**, las iniciales de Primary Key. Pero esto es optativo, el modelo relacional no obliga a elegir una clave principal para cada relación, sino tan sólo a la existencia de al menos una clave candidata.

La inexistencia de claves candidatas implica que la relación no cumple todas las normas para ser parte de una base de datos relacional, ya que la no existencia de claves implica la repetición de tuplas.

En general, si no existe un candidato claro para la clave principal, crearemos una columna específica con ese propósito.

Veamos cómo aplicar esta regla usando un ejemplo. En este caso trataremos de guardar datos relacionados con la administración de un hotel.

Planteemos, por ejemplo, este esquema de relación:

```
Ocupación(No_cliente, Nombre_cliente, No_habitación,
           precio_noche, tipo_habitación, fecha_entrada)
```

Lo primero que tenemos que hacer es buscar las posibles claves candidatas. En este caso sólo existe una posibilidad:

```
(No_habitación, fecha_entrada)
```

Recordemos que cualquier clave candidata debe identificar de forma unívoca una clave completa. En este caso, la clave completa es la ocupación de una habitación, que se define por dos parámetros: la habitación y la fecha de la ocupación.

Es decir, dos ocupaciones son diferentes si cambian cualquiera de estos parámetros. La misma persona puede ocupar varias habitaciones al mismo tiempo o la misma habitación durante varios días o en

diferentes periodos de tiempo. Lo que no es posible es que varias personas ocupen la misma habitación al mismo tiempo (salvo que se trate de un acompañante, pero en ese caso, sólo una de las personas es la titular de la ocupación).

El siguiente paso consiste en buscar columnas que no aporten información directa sobre la clave completa: la ocupación. En este caso el precio por noche de la habitación y el tipo de habitación (es decir, si es doble o sencilla), no aportan información sobre la clave principal. En realidad, estos datos no son atributos de la ocupación de la habitación, sino de la propia habitación.

El número de cliente y su nombre aportan datos sobre la ocupación, aunque no formen parte de la clave completa.

Expresado en forma de dependencias se ve muy claro:

```
(No_habitación, fecha_entrada) -> No_cliente
(No_habitación, fecha_entrada) -> Nombre_cliente
No_habitación -> precio_noche
No_habitación -> tipo_habitación
```

El último paso consiste en extraer los atributos que no forman parte de la clave a otra relación. En nuestro ejemplo tendremos dos relaciones: una para las ocupaciones y otra para las habitaciones:

```
Ocupación(No_cliente, Nombre_cliente, No_habitación, fecha_entrada(PK))
Habitación(No_habitación, precio_noche, tipo_habitación)
```

La segunda tabla tiene una única clave candidata, que es el número de habitación. El resto de los atributos no forman parte de la clave completa (la habitación), pero aportan información sólo y exclusivamente sobre ella.

Estas dos relaciones están interrelacionadas por la clave de habitación. Para facilitar el establecimiento de esta interrelación elegiremos la clave candidata de la habitación en clave principal. El mismo atributo en la relación de ocupaciones es, por lo tanto, una clave foránea:

```
Ocupación(No_cliente, Nombre_cliente, No_habitación, fecha_entrada(PK))
Habitación(No_habitación(PK), precio_noche, tipo_habitación)
```

Como norma general debemos volver a aplicar la primera y segunda forma normal a estas nuevas tablas. La primera sólo en el caso de que hallamos añadido nuevas columnas, la segunda siempre.

La interrelación que hemos establecido es del tipo uno a muchos, podemos elegir una clave de habitación muchas veces, tantas como veces se ocupe esa habitación.

## Dependencia funcional transitiva

**Definición:** Supongamos que tenemos una relación con tres conjuntos de atributos: X, Y y Z, y las siguientes dependencias  $X \rightarrow Y$ ,  $Y \rightarrow Z$ ,  $Y \not\rightarrow X$ . Es decir X determina Y e Y determina Z, pero Y no determina X. En ese caso, decimos que Z tiene dependencia transitiva con respecto a X, a través de Y.

Intentaremos aclarar este concepto tan teórico con un ejemplo. Si tenemos esta relación:

**Ciudades**(ciudad, población, superficie, renta, país, continente)

Los atributos como *población*, *superficie* o *renta* tienen dependencia funcional de *ciudad*, así que de momento no nos preocupan.

En esta relación podemos encontrar también las siguientes dependencias:

$ciudad \rightarrow país$ ,  $país \rightarrow continente$ . Además,  $país \not\rightarrow ciudad$ . Es decir, cada *ciudad* pertenece a un *país* y cada *país* a un *continente*, pero en cada *país* puede haber muchas *ciudades*. En este caso *continente* tiene una dependencia funcional transitiva con respecto a *ciudad*, a través de *país*. Es decir, cada *ciudad* está en un *país*, pero también en un *continente*. (¡Ojo! las dependencias transitivas no son siempre tan evidentes :-).

## Tercera forma normal 3FN

La tercera forma normal consiste en eliminar las dependencias transitivas.

**Definición:** Una base de datos está en 3FN si está en 2FN y además todas las columnas que no sean claves dependen de la clave completa de forma no transitiva.

Pero esto es una definición demasiado teórica. En la práctica significa que se debe eliminar cualquier relación que permita llegar a un mismo dato de dos o más formas diferentes.

Tomemos el ejemplo que usamos para ilustrar las dependencias funcionales transitivas. Tenemos una tabla donde se almacenen datos relativos a ciudades, y una de las columnas sea el país y otra el continente al que pertenecen. Por ejemplo:

```
Ciudades(ID_ciudad(PK), Nombre, población, superficie, renta, país, continente)
```

Un conjunto de datos podría ser el siguiente:

<b>Ciudades</b>						
<u>ID_ciudad</u>	<u>Nombre</u>	<u>población</u>	<u>superficie</u>	<u>renta</u>	<u>país</u>	<u>continente</u>
1	Paris	6000000	15	1800	Francia	Europa
2	Lion	3500000	9	1600	Francia	Europa
3	Berlin	7500000	16	1900	Alemania	Europa
4	Pekin	19000000	36	550	China	Asia
5	Bonn	6000000	12	1900	Alemania	Europa

Podemos ver que para cada aparición de un determinado país, el continente siempre es el mismo. Es decir, existe una redundancia de datos, y por lo tanto, un peligro de integridad.

Existe una relación entre país y continente, y ninguna de ellas es clave candidata. Por lo tanto, si queremos que esta table sea 3FN debemos separar esa columna:

```
Ciudades(ID_ciudad(PK), Nombre, población, superficie, renta, nombre_pais)
Países(nombre_pais(PK), nombre_continente)
```

<b>Ciudades</b>					
<u>ID_ciudad</u>	<u>Nombre</u>	<u>población</u>	<u>superficie</u>	<u>renta</u>	<u>país</u>
1	Paris	6000000	15	1800	Francia
2	Lion	3500000	9	1600	Francia
3	Berlin	7500000	16	1900	Alemania
4	Pekin	19000000	36	550	China
5	Bonn	6000000	12	1900	Alemania

<b>Países</b>	
<u>país</u>	<u>continente</u>
Francia	Europa
Alemania	Europa
China	Asia

Esta separación tendría más sentido si la tabla de países contuviese más información, tal como está no tiene mucho sentido separar estas tablas, aunque efectivamente, se evita redundancia.

## Forma Normal de Boyce y Codd (FNBC)

**Definición:** Una relación está en FNBC si cualquier atributo sólo facilita información sobre claves candidatas, y no sobre atributos que no formen parte de ninguna clave candidata.

Esto significa que no deben existir interrelaciones entre atributos fuera de las claves candidatas.

Para ilustrar esta forma normal volvamos a uno de nuestros ejemplos anteriores, el de las ocupaciones de habitaciones de un hotel.

```
Ocupación(No_cliente, Nombre_cliente, No_habitación, fecha_entrada)
Habitación(No_habitación(PK), precio_noche, tipo_habitación)
```

En la primera relación los atributos `No_cliente` y `Nombre_cliente` sólo proporcionan información entre ellos mutuamente, pero ninguno de ellos es una clave candidata.

Intuitivamente ya habremos visto que esta estructura puede producir redundancia, sobre todo en el caso de clientes habituales, donde se repetirá la misma información cada vez que el mismo cliente se aloje en el hotel.

La solución, como siempre, es simple, y consiste en separar esta relación en dos diferentes:

```
Ocupación(No_cliente, No_habitación, fecha_entrada)
Cliente(No_cliente(PK), Nombre_cliente)
Habitación(No_habitación(PK), precio_noche, tipo_habitación)
```

## Atributos multivaluados

**Definición:** se dice que un atributo es multivaluado cuando para una misma entidad puede tomar varios valores diferentes, con independencia de los valores que puedan tomar el resto de los atributos.

Se representa como  $X \twoheadrightarrow Y$ , y se lee como *X multidetermina Y*.

Un ejemplo claro es una relación donde almacenemos contactos y números de teléfono:

```
Agenda(nombre, fecha_nacimiento, estado_civil, teléfono)
```

Para cada *nombre* de la agenda tendremos, en general, varios números de *teléfono*, es decir, que *nombre* multidetermina *teléfono*:  $\text{nombre} \twoheadrightarrow \text{teléfono}$ .

Además, *nombre* determina funcionalmente otros atributos, como la *fecha\_nacimiento* o *estado\_civil*.

Por otra parte, la clave candidata no es el nombre, ya que debe ser unívoca, por lo tanto debe ser una combinación de nombre y teléfono.

En esta relación tenemos las siguientes dependencias:

- $\text{nombre} \rightarrow \text{fecha\_nacimiento}$
- $\text{nombre} \rightarrow \text{estado\_civil}$
- $\text{nombre} \twoheadrightarrow \text{teléfono}$ , o lo que es lo mismo  $(\text{nombre}, \text{teléfono}) \rightarrow \text{teléfono}$

Es decir, la dependencia multivaluada se convierte, de hecho, en una dependencia funcional trivial.

Este tipo de atributos implica redundancia ya que el resto de los atributos se repiten tantas veces como valores diferentes tenga el atributo multivaluado:

<b>Agenda</b>	<u>nombre</u>	<u>fecha_nacimiento</u>	<u>estado_civil</u>	<u>teléfono</u>
Mengano	15/12/1985	soltero	12322132	
Fulano	13/02/1960	casado	13321232	
Fulano	13/02/1960	casado	25565445	
Fulano	13/02/1960	casado	36635363	
Tulana	24/06/1975	soltera	45665456	

Siempre podemos evitar el problema de los atributos multivaluados separándolos en relaciones distintas. En el ejemplo anterior, podemos crear una segunda relación que contenga el nombre y el teléfono:

```
Agenda(nombre(PK), fecha_nacimiento, estado_civil)
Teléfonos(nombre, teléfono(PK))
```

Para los datos anteriores, las tablas quedarían así:

### Agenda

<u>nombre</u>	<u>fecha_nacimiento</u>	<u>estado_civil</u>
Mengano	15/12/1985	soltero
Fulano	13/02/1960	casado
Tulana	24/06/1975	soltera

### Teléfonos

<u>nombre</u>	<u>teléfono</u>
Mengano	12322132
Fulano	13321232
Fulano	25565445
Fulano	36635363
Tulana	45665456

## Dependencias multivaluadas

Si existe más de un atributo multivaluado es cuando se presentan dependencias multivaluadas.

**Definición:** en una relación con los atributos X, Y y Z existe una dependencia multivaluada de Y con respecto a X si los posibles valores de Y para un par de valores de X y Z dependen únicamente del valor de X.

Supongamos que en nuestra relación anterior de **Agenda** añadimos otro atributo para guardar direcciones de correo electrónico. Se trata, por supuesto, de otro atributo multivaluado, ya que cada persona puede tener más de una dirección de correo, y este atributo es independiente del número de teléfono:

```
Agenda(nombre, fecha_nacimiento, estado_civil, teléfono, correo)
```

Ahora surgen los problemas, supongamos que nuestro amigo "Fulano", además de los tres números de teléfono, dispone de dos direcciones de correo. ¿Cómo almacenaremos la información relativa a estos datos? Tenemos muchas opciones:

**Agenda**

<u>nombre</u>	<u>fecha_nacimiento</u>	<u>estado_civil</u>	<u>teléfono</u>	<u>correo</u>
Fulano	13/02/1960	casado	13321232	fulano@sucasa.eko
Fulano	13/02/1960	casado	25565445	fulano@sutrabajo.aka
Fulano	13/02/1960	casado	36635363	fulano@sucasa.eko

Si optamos por crear tres tuplas, ya que hay tres teléfonos, y emparejar cada dirección con un teléfono, en la tercera tupla estaremos obligados a repetir una dirección. Otra opción sería usar un *NULL* en esa tercera tupla.

**Agenda**

<u>nombre</u>	<u>fecha_nacimiento</u>	<u>estado_civil</u>	<u>teléfono</u>	<u>correo</u>
Fulano	13/02/1960	casado	13321232	fulano@sucasa.eko
Fulano	13/02/1960	casado	25565445	fulano@sutrabajo.aka
Fulano	13/02/1960	casado	36635363	NULL

Pero estas opciones ocultan el hecho de que ambos atributos son multivaluados e independientes entre si. Podría parecer que existe una relación entre los números y las direcciones de correo.

Intentemos pensar en otras soluciones:

**Agenda**

<u>nombre</u>	<u>fecha_nacimiento</u>	<u>estado_civil</u>	<u>teléfono</u>	<u>correo</u>
Fulano	13/02/1960	casado	13321232	fulano@sucasa.eko
Fulano	13/02/1960	casado	25565445	fulano@sucasa.aka
Fulano	13/02/1960	casado	36635363	fulano@sucasa.eko
Fulano	13/02/1960	casado	13321232	fulano@sutrabajo.eko
Fulano	13/02/1960	casado	25565445	fulano@sutrabajo.aka
Fulano	13/02/1960	casado	36635363	fulano@sutrabajo.eko

**Agenda**

<u>nombre</u>	<u>fecha_nacimiento</u>	<u>estado_civil</u>	<u>teléfono</u>	<u>correo</u>
Fulano	13/02/1960	casado	13321232	NULL
Fulano	13/02/1960	casado	25565445	NULL
Fulano	13/02/1960	casado	36635363	NULL
Fulano	13/02/1960	casado	NULL	fulano@sutrabajo.eko
Fulano	13/02/1960	casado	NULL	fulano@sucasa.eko

Ahora está claro que los atributos son independientes, pero el precio es crear más tuplas para guardar la misma información, es decir, mayor redundancia.

Pero no sólo eso. Las operaciones de inserción de nuevos datos, corrección o borrado se complican. Ninguna de esas operaciones se puede hacer modificando sólo una tupla, y cuando eso sucede es posible que se produzcan inconsistencias.

## Cuarta forma normal (4FN)

La cuarta forma normal tiene por objetivo eliminar las dependencias multivaluadas.

**Definición:** Una relación está en 4NF si y sólo si, en cada dependencia multivaluada  $X \twoheadrightarrow Y$  no trivial,  $X$  es clave candidata.

Una dependencia multivaluada  $A \twoheadrightarrow B$  es trivial cuando  $B$  es parte de  $A$ . Esto sucede cuando  $A$  es un conjunto de atributos, y  $B$  es un subconjunto de  $A$ .

Tomemos por ejemplo la tabla de Agenda, pero dejando sólo los atributos multivaluados:

```
Agenda(nombre, teléfono, correo)
```

Lo primero que debemos hacer es buscar las claves y las dependencias. Recordemos que las claves candidatas deben identificar de forma unívoca cada tupla. De modo que estamos obligados a usar los tres atributos para formar la clave candidata.

Pero las dependencias que tenemos son:

- nombre  $\twoheadrightarrow$  teléfono
- nombre  $\twoheadrightarrow$  correo

Y *nombre* no es clave candidata de esta relación.

Resumiendo, debemos separar esta relación en varias (tantas como atributos multivaluados tenga).

```
Teléfonos(nombre, teléfono)
Correos(nombre, correo)
```

Ahora en las dos relaciones se cumple la cuarta forma normal.

## Quinta forma normal (5FN)

Existe una quinta forma normal, pero no la veremos en este curso. Sirve para eliminar dependencias de proyección o reunión, que raramente se encuentran en las bases de datos que probablemente manejaremos. Si tienes interés en saber más sobre este tema, consulta la bibliografía.

### Ejemplo 1

Aplicaremos ahora la normalización a las relaciones que obtuvimos en el capítulo anterior.

En el caso del primer ejemplo, para almacenar información sobre estaciones meteorológicas y las muestras tomadas por ellas, habíamos llegado a esta estructura:

```
Estación(Identificador, Latitud, Longitud, Altitud)
Muestra(IdentificadorEstacion, Fecha, Temperatura mínima, Temperatura
máxima,
    Precipitaciones, Humedad mínima, Humedad máxima, Velocidad del viento
mínima,
    Velocidad del viento máxima)
```

### Primera forma normal

Esta forma nos dice que todos los atributos deben ser atómicos.

Ya comentamos antes que este criterio es en cierto modo relativo, lo que desde un punto de vista puede ser atómico, puede no serlo desde otro.

En lo que respecta a la relación *Estación*, el Identificador y la Altitud son claramente atómicos. Sin embargo, la Latitud y Longitud pueden considerarse desde dos puntos de vista. En uno son coordenadas (de hecho, podríamos haber considerado la posición como atómica, y fundir ambos atributos en uno). A pesar de que ambos valores se expresen en grados, minutos y segundos, más una orientación, norte, sur, este u oeste, puede hacernos pensar que podemos dividir ambos atributos en partes más simples.

Esta es, desde luego, una opción. Pero todo depende del uso que le vayamos a dar a estos datos. Para nuestra aplicación podemos considerar como atómicos estos dos atributos tal como los hemos definido.

Para la relación *Muestras* todos los atributos seleccionados son atómicos.

## Segunda forma normal

Para que una base de datos sea 2FN primero debe ser 1FN, y además todas las columnas que formen parte de una clave candidata deben aportar información sobre la clave completa.

Para la relación *Estación* existen dos claves candidatas: identificador y la formada por la combinación de Latitud y Longitud.

Hay que tener en cuenta que hemos creado el atributo Identificador sólo para ser usado como clave principal. Las dependencias son:

```
Identificador -> (Latitud, Longitud)
Identificador -> Altitud
```

En estas dependencias, las claves candidatas Identificador y (Latitud, Longitud) son equivalentes y, por lo tanto, intercambiables.

Esta relación se ajusta a la segunda forma normal, veamos la de *Muestras*.

En esta relación la clave principal es la combinación de (Identificador, Fecha), y el resto de los atributos son dependencias funcionales de esta clave. Por lo tanto, también esta relación está en 2FN.

## Tercera forma normal

Una base de datos está en 3FN si está en 2FN y además todas las columnas que no sean claves dependen de la clave completa de forma no transitiva.

No existen dependencias transitivas, de modo que podemos afirmar que nuestra base de datos está en 3FN.

## Forma normal de Boyce/Codd

Una relación está en FNBC si cualquier atributo sólo facilita información sobre claves candidatas, y no sobre atributos que no formen parte de ninguna clave candidata.

Tampoco existen atributos que den información sobre otros atributos que no sean o formen parte de claves candidatas.

## Cuarta forma normal

Esta forma se refiere a atributos multivaluados, que no existen en nuestras relaciones, por lo tanto, también podemos considerar que nuestra base de datos está en 4FN.

## Ejemplo 2

Nuestro segundo ejemplo se modela una biblioteca, y su esquema de relaciones final es este:

```

Libro(ClaveLibro, Título, Idioma, Formato, Categoría, ClaveEditorial)
Tema(ClaveTema, Nombre)
Autor(ClaveAutor, Nombre)
Editorial(ClaveEditorial, Nombre, Dirección, Teléfono)
Ejemplar(ClaveLibro, NúmeroOrden, Edición, Ubicación)
Socio(ClaveSocio, Nombre, Dirección, Teléfono, Categoría)
Préstamo(ClaveSocio, ClaveLibro, NúmeroOrden, Fecha_préstamo,
  Fecha_devolución, Notas)
Trata_sobre(ClaveLibro, ClaveTema)
Escrito_por(ClaveLibro, ClaveAutor)

```

Los ejemplos que estamos siguiendo desde el capítulo 2 demuestran que un buen diseño conceptual sirve para diseñar bases de datos relacionales libres de redundancias, y generalmente, la normalización no afecta a su estructura.

Este ha sido el caso del primer ejemplo, y como veremos, también del segundo.

## Primera forma normal

Tenemos, desde luego, algunos atributos que podemos considerar no atómicos, como el nombre del autor, la dirección de la editorial, el nombre del socio y su dirección. Como siempre, dividir estos atributos en otros es una decisión de diseño, que dependerá del uso que se le vaya a dar a estos datos. En nuestro caso, seguramente sea suficiente dejarlos tal como están, pero dividir algunos de ellos no afecta demasiado a las relaciones.

## Segunda forma normal

Para que una base de datos sea 2FN primero debe ser 1FN, y además todas las columnas que formen parte de una clave candidata deben aportar información sobre la clave completa.

En el caso de *Libro*, la única clave candidata es ClaveLibro. Todos los demás valores son repetibles, pueden existir libros con el mismo título y de la misma editorial editados en el mismo formato e idioma y que englobemos en la misma categoría. Es decir, no existe ningún otro atributo o conjunto de atributos

que puedan identificar un libro de forma unívoca.

Se pueden dar casos especiales, como el del mismo libro escrito en diferentes idiomas. En ese caso la clave será diferente, de modo que los consideraremos como libros distintos. Lo mismo pasa si el mismo libro aparece en varios formatos, o ha sido editado por distintas editoriales.

Es decir, todos los atributos son dependencias funcionales de ClaveLibro.

Con *Tema* y *Autor* no hay dudas, sólo tienen dos atributos, y uno de ellos ha sido creado específicamente para ser usado como clave.

Los tres atributos de *Editorial* también tienen dependencia funcional de ClaveEditorial.

Y lo mismo cabe decir para las entidades *Ejemplar*, *Socio* y *Préstamo*.

En cuanto a las relaciones que almacenan interrelaciones, la clave es el conjunto de todos los atributos, de modo que todas las dependencias son funcionales y triviales.

### Tercera forma normal

Una base de datos está en 3FN si está en 2FN y además todas las columnas que no sean claves dependen de la clave completa de forma no transitiva.

En *Libro* no hay ningún atributo que tenga dependencia funcional de otro atributo que no sea la clave principal. Todos los atributos definen a la entidad *Libro* y a ninguna otra.

Las entidades con sólo dos atributos no pueden tener dependencias transitivas, como *Tema* o *Autor*.

Con *Editorial* tampoco existen, todos los atributos dependen exclusivamente de la clave principal.

En el caso del *Ejemplar* tampoco hay una correspondencia entre ubicación y edición. O al menos no podemos afirmar que exista una norma universal para esta correspondencia. Es posible que todas las primeras ediciones se guarden en el mismo sitio, pero esto no puede ser una condición de diseño para la base de datos.

Y para *Préstamo* los tres atributos que no forman parte de la clave candidata se refieren sólo a la entidad *Préstamo*.

### Forma normal de Boyce/Codd

Una relación está en FNBC si cualquier atributo sólo facilita información sobre claves candidatas, y no

sobre atributos que no formen parte de ninguna clave candidata.

Tampoco existen atributos que den información sobre otros atributos que no sean o formen parte de claves candidatas.

## Cuarta forma normal

No hay atributos multivaluados. O mejor dicho, los que había ya se convirtieron en entidades cuando diseñamos el modelo E-R.

### Ejemplo 3

La normalización será mucho más útil cuando nuestro diseño arranque directamente en el modelo relacional, es decir, cuando no arranquemos de un modelo E-R. Si no somos cuidadosos podemos introducir relaciones con más de una entidad, dependencias transitivas o atributos multivaluados.

## 5 Tipos de columnas

Una vez que hemos decidido qué información debemos almacenar, hemos normalizado nuestras tablas y hemos creado claves principales, el siguiente paso consiste en elegir el tipo adecuado para cada atributo.

En **MySQL** existen bastantes tipos diferentes disponibles, de modo que será mejor que los agrupemos por categorías: de caracteres, enteros, de coma flotante, tiempos, bloques, enumerados y conjuntos.

### Tipos de datos de cadenas de caracteres

#### CHAR

```
CHAR
```

Es un sinónimo de CHAR(1), y puede contener un único carácter.

#### CHAR()

```
[NATIONAL] CHAR(M) [BINARY | ASCII | UNICODE]
```

Contiene una cadena de longitud constante. Para mantener la longitud de la cadena, se rellena a la derecha con espacios. Estos espacios se eliminan al recuperar el valor.

Los valores válidos para M son de 0 a 255, y de 1 a 255 para versiones de MySQL previas a 3.23.

Si no se especifica la palabra clave *BINARY* estos valores se ordenan y comparan sin distinguir mayúsculas y minúsculas.

**CHAR** es un alias para **CHARACTER**.

#### VARCHAR()

```
[NATIONAL] VARCHAR(M) [BINARY]
```

Contiene una cadena de longitud variable. Los valores válidos para M son de 0 a 255, y de 1 a 255 en versiones de MySQL anteriores a 4.0.2.

Los espacios al final se eliminan.

Si no se especifica la palabra clave *BINARY* estos valores se ordenan y comparan sin distinguir mayúsculas y minúsculas.

**VARCHAR** es un alias para **CHARACTER VARYING**.

## Tipos de datos enteros

### TINYINT

```
TINYINT[(M)] [UNSIGNED] [ZEROFILL]
```

Contiene un valor entero muy pequeño. El rango con signo es entre -128 y 127. El rango sin signo, de 0 a 255.

**BIT**  
**BOOL**  
**BOOLEAN**

Todos son sinónimos de **TINYINT(1)**.

### SMALLINT

```
SMALLINT[(M)] [UNSIGNED] [ZEROFILL]
```

Contiene un entero corto. El rango con signo es de -32768 a 32767. El rango sin signo, de 0 a 65535.

### MEDIUMINT

```
MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]
```

Contiene un entero de tamaño medio, el rango con signo está entre -8388608 y 8388607. El rango sin signo, entre 0 y 16777215.

## INT

```
INT[(M)] [UNSIGNED] [ZEROFILL]
```

Contiene un entero de tamaño normal. El rango con signo está entre -2147483648 y 2147483647. El rango sin signo, entre 0 y 4294967295.

## INTEGER

```
INTEGER[(M)] [UNSIGNED] [ZEROFILL]
```

Es sinónimo de **INT**.

## BIGINT

```
BIGINT[(M)] [UNSIGNED] [ZEROFILL]
```

Contiene un entero grande. El rango con signo es de -9223372036854775808 a 9223372036854775807. El rango sin signo, de 0 a 18446744073709551615.

## Tipos de datos en coma flotante

### FLOAT

```
FLOAT(precision) [UNSIGNED] [ZEROFILL]
```

Contiene un número en coma flotante. *precision* puede ser menor o igual que 24 para números de precisión sencilla y entre 25 y 53 para números en coma flotante de doble precisión. Estos tipos son idénticos que los tipos **FLOAT** y **DOUBLE** descritos a continuación. **FLOAT(X)** tiene el mismo rango que los tipos **FLOAT** y **DOUBLE** correspondientes, pero el tamaño mostrado y el número de decimales quedan indefinidos.

## FLOAT()

```

FLOAT[(M,D)] [UNSIGNED] [ZEROFILL]

```

Contiene un número en coma flotante pequeño (de precisión sencilla). Los valores permitidos son entre  $-3.402823466E+38$  y  $-1.175494351E-38$ , 0, y entre  $1.175494351E-38$  y  $3.402823466E+38$ . Si se especifica el modificador *UNSIGNED*, los valores negativos no se permiten.

El valor *M* es la anchura a mostrar y *D* es el número de decimales. Si se usa sin argumentos o si se usa **FLOAT(X)**, donde *X* sea menor o igual que 24, se sigue definiendo un valor en coma flotante de precisión sencilla.

## DOUBLE

```

DOUBLE[(M,D)] [UNSIGNED] [ZEROFILL]

```

Contiene un número en coma flotante de tamaño normal (precisión doble). Los valores permitidos están entre  $-1.7976931348623157E+308$  y  $-2.2250738585072014E-308$ , 0, y entre  $2.2250738585072014E-308$  y  $1.7976931348623157E+308$ . Si se especifica el modificador *UNSIGNED*, no se permiten los valores negativos.

El valor *M* es la anchura a mostrar y *D* es el número de decimales. Si se usa sin argumentos o si se usa **DOUBLE(X)**, donde *X* esté entre 25 y 53, se sigue definiendo un valor en coma flotante de doble precisión.

## DOUBLE PRECISION REAL

```

DOUBLE PRECISION[(M,D)] [UNSIGNED] [ZEROFILL]
REAL[(M,D)] [UNSIGNED] [ZEROFILL]

```

Ambos son sinónimos de **DOUBLE**.

## DECIMAL

```

DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]

```

Contiene un número en coma flotante sin empaquetar. Se comporta igual que una columna **CHAR**: "sin empaquetar" significa que se almacena como una cadena, usando un carácter para cada dígito del valor. El punto decimal y el signo '-' para valores negativos, no se cuentan en M (pero el espacio para estos se reserva). Si D es 0, los valores no tendrán punto decimal ni decimales.

El rango de los valores **DECIMAL** es el mismo que para **DOUBLE**, pero el rango actual para una columna **DECIMAL** dada está restringido por la elección de los valores M y D.

Si se especifica el modificador *UNSIGNED*, los valores negativos no están permitidos.

Si se omite D, el valor por defecto es 0. Si se omite M, el valor por defecto es 10.

## DEC NUMERIC FIXED

```
DEC[(M[,D])] [UNSIGNED] [ZEROFILL]  
NUMERIC[(M[,D])] [UNSIGNED] [ZEROFILL]  
FIXED[(M[,D])] [UNSIGNED] [ZEROFILL]
```

Todos ellos son sinónimos de **DECIMAL**.

## Tipos de datos para tiempos

### DATE

```
DATE
```

Contiene una fecha. El rango soportado está entre '1000-01-01' y '9999-12-31'. MySQL muestra los valores **DATE** con el formato 'AAAA-MM-DD', pero es posible asignar valores a columnas de este tipo usando tanto números como cadenas.

### DATETIME

```
DATETIME
```

Contiene una combinación de fecha y hora. El rango soportado está entre '1000-01-01 00:00:00' y '9999-12-31 23:59:59'. MySQL muestra los valores **DATETIME** con el formato 'AAAA-MM-DD HH:MM:SS', pero es posible asignar valores a columnas de este tipo usando tanto cadenas como números.

## TIMESTAMP

```
TIMESTAMP [ (M) ]
```

Contiene un valor del tipo timestamp. El rango está entre '1970-01-01 00:00:00' y algún momento del año 2037.

Hasta MySQL 4.0 los valores **TIMESTAMP** se mostraban como AAAAMMDDHHMMSS, AAMMDDHHMMSS, AAAAMMDD o AAMMDD, dependiendo del si el valor de M es 14 (o se omite), 12, 8 o 6, pero está permitido asignar valores a columnas **TIMESTAMP** usando tanto cadenas como números.

Desde MySQL 4.1, **TIMESTAMP** se devuelve como una cadena con el formato 'AAAA-MM-DD HH:MM:SS'. Para convertir este valor a un número, bastará con sumar el valor 0. Ya no se soportan distintas longitudes para estas columnas.

Se puede asignar fácilmente la fecha y hora actual a uno de estas columnas asignando el valor NULL.

El argumento M afecta sólo al modo en que se visualiza la columna **TIMESTAMP**. Los valores siempre se almacenan usando cuatro bytes. Además, los valores de columnas **TIMESTAMP(M)**, cuando M es 8 ó 14 se devuelven como números, mientras que para el resto de valores se devuelven como cadenas.

## TIME

```
TIME
```

Una hora. El rango está entre '-838:59:59' y '838:59:59'. MySQL muestra los valores **TIME** en el formato 'HH:MM:SS', pero permite asignar valores a columnas **TIME** usando tanto cadenas como números.

## YEAR

```
YEAR[ ( 2 | 4 ) ]
```

Contiene un año en formato de 2 ó 4 dígitos (por defecto es 4). Los valores válidos son entre 1901 y 2155, y 0000 en el formato de 4 dígitos. Y entre 1970-2069 si se usa el formato de 3 dígitos (70-69).

MySQL muestra los valores **YEAR** usando el formato AAAA, pero permite asignar valores a una columna **YEAR** usando tanto cadenas como números.

## Tipos de datos para datos sin tipo o grandes bloques de datos

**TINYBLOB**  
**TINYTEXT**

```
TINYBLOB  
TINYTEXT
```

Contiene una columna **BLOB** o **TEXT** con una longitud máxima de 255 caracteres ( $2^8 - 1$ ).

**BLOB**  
**TEXT**

```
BLOB  
TEXT
```

Contiene una columna **BLOB** o **TEXT** con una longitud máxima de 65535 caracteres ( $2^{16} - 1$ ).

**MEDIUMBLOB**  
**MEDIUMTEXT**

```
MEDIUMBLOB  
MEDIUMTEXT
```

Contiene una columna **BLOB** o **TEXT** con una longitud máxima de 16777215 caracteres ( $2^{24} - 1$ ).

**LOBLOB**

## LONGTEXT

```

LONGBLOB
LONGTEXT

```

Contiene una columna **BLOB** o **TEXT** con una longitud máxima de 4294967298 caracteres ( $2^{32} - 1$ ).

## Tipos enumerados y conjuntos

### ENUM

```

ENUM('valor1', 'valor2', ...)

```

Contiene un enumerado. Un objeto de tipo cadena que puede tener un único valor, entre una lista de valores 'valor1', 'valor2', ..., NULL o el valor especial de error "". Un **ENUM** puede tener un máximo de 65535 valores diferentes.

### SET

```

SET('valor1', 'valor2', ...)

```

Contiene un conjunto. Un objeto de tipo cadena que puede tener cero o más valores, cada uno de los cuales debe estar entre una lista de valores 'valor1', 'valor2', ...

Un conjunto puede tener un máximo de 64 miembros.

## Ejemplo 1

El siguiente paso del diseño nos obliga a elegir tipos para cada atributo de cada relación. Veamos cómo lo hacemos para los ejemplos que hacemos en cada capítulo.

Para el primer ejemplo teníamos el siguiente esquema:

```

Estación(Identificador, Latitud, Longitud, Altitud)
Muestra(IdentificadorEstación, Fecha, Temperatura mínima, Temperatura

```

```
máxima,
  Precipitaciones, Humedad mínima, Humedad máxima, Velocidad del viento
mínima,
  Velocidad del viento máxima)
```

En **MySQL** es importante, aunque no obligatorio, usar valores enteros como claves principales, ya que las optimizaciones que proporcionan un comportamiento mucho mejor para claves enteras.

Vamos a elegir el tipo de cada atributo, uno por uno:

## Relación Estación

**Identificador:** podríamos pensar en un entero corto o medio, aunque no tenemos datos sobre el número de estaciones que debemos manejar, no es probable que sean más de 16 millones. Este dato habría que incluirlo en la documentación, pero supongamos que con **MEDIUMINT UNSIGNED** es suficiente.

**Latitud:** las latitudes se expresan en grados, minutos y segundos, al norte o sur del ecuador. Los valores están entre 'N90°00'00.000"' y 'S90°00'00.000"'. Además, los segundos, dependiendo de la precisión de la posición que almacenemos, pueden tener hasta tres decimales. Para este tipo de datos tenemos dos opciones. La primera es la que comentamos en el capítulo anterior: no considerar este valor como atómico, y guardar tres números y la orientación N/S como atributos separados. Si optamos por la segunda, deberemos usar una cadena, que tendrá como máximo 14 caracteres. El tipo puede ser **CHAR(14)** o **VARCHAR(14)**.

**Longitud:** las longitudes se almacenan en un formato parecido, pero la orientación es este/oeste, y el valor de grados varía entre 0 y 180, es decir, que necesitamos un carácter más: **CHAR(15)** o **VARCHAR(15)**.

**Altitud:** es un número entero, que puede ser negativo si la estación está situada en una depresión, y como máximo a unos 8000 metros (si alguien se atreve a colocar una estación en el Everest. Esto significa que con un **MEDIUMINT** tenemos más que suficiente.

## Relación Muestra

**IdentificadorEstación:** se trata de una clave foránea, de modo que debe ser del mismo tipo que la clave primaria de la que procede: **MEDIUMINT UNSIGNED**.

**Fecha:** sólo necesitamos almacenar una fecha, de modo que con el tipo **DATE** será más que suficiente.

**Temperatura mínima:** las temperaturas ambientes en grados centígrados (Celsius) se pueden

almacenar en un entero muy pequeño, TINYINT, que permite un rango entre -128 y 127. Salvo que se sitúen estaciones en volcanes, no es probable que se salga de estos rangos. Recordemos que las muestras tienen aplicaciones meteorológicas.

**Temperatura máxima:** lo mismo que para la temperatura mínima: TINYINT.

**Precipitaciones:** personalmente, ignoro cuánto puede llegar a llover en un día, pero supongamos que 255 litros por metro cuadrado sea una cantidad que se puede superar. En ese caso estamos obligados a usar el siguiente rango: SMALLINT UNSIGNED, que nos permite almacenar números hasta 65535.

**Humedad mínima:** las humedades se miden en porcentajes, el valor está acotado entre 0 y 100, de nuevo nos bastará con un TINYINT, nos da lo mismo con o sin signo, pero usaremos el UNSIGNED.

**Humedad máxima:** También TINYINT UNSIGNED.

**Velocidad del viento mínima:** también estamos en valores siempre positivos, aunque es posible que se superen los 255 Km/h, de modo que, para estar seguros, usaremos SMALLINT UNSIGNED.

**Velocidad del viento máxima:** también usaremos SMALLINT UNSIGNED.

## Ejemplo 2

Para el segundo ejemplo partimos del siguiente esquema:

```

Libro(ClaveLibro, Título, Idioma, Formato, Categoría, ClaveEditorial)
Tema(ClaveTema, Nombre)
Autor(ClaveAutor, Nombre)
Editorial(ClaveEditorial, Nombre, Dirección, Teléfono)
Ejemplar(ClaveLibro, NúmeroOrden, Edición, Ubicación)
Socio(ClaveSocio, Nombre, Dirección, Teléfono, Categoría)
Préstamo(ClaveSocio, ClaveLibro, NúmeroOrden, Fecha_préstamo,
  Fecha_devolución, Notas)
Trata_sobre(ClaveLibro, ClaveTema)
Escrito_por(ClaveLibro, ClaveAutor)
  
```

## Relación *Libro*

**ClaveLibro:** como clave principal que es, este atributo debe ser de tipo entero. Una biblioteca puede tener muchos libros, de modo que podemos usar el tipo INT.

**Título:** para este atributo usaremos una cadena de caracteres, la longitud es algo difícil de decidir, pero como primera aproximación podemos usar un VARCHAR(60).

**Idioma:** usaremos una cadena de caracteres, por ejemplo, VARCHAR(15).

**Formato:** se trata de otra palabra, por ejemplo, VARCHAR(15).

**Categoría:** recordemos que para este atributo usábamos una letra, por lo tanto usaremos el tipo CHAR.

**ClaveEditorial:** es una clave foránea, y por lo tanto, el tipo debe ser el mismo que para el atributo 'ClaveEditorial' de la tabla 'Editorial', que será SMALLINT.

### **Relación Tema**

**ClaveTema:** el número de temas puede ser relativamente pequeño, con un SMALLINT será más que suficiente.

**Nombre:** otra cadena, por ejemplo, VARCHAR(40).

### **Relación Autor**

**ClaveAutor:** usaremos, al igual que con los libros, el tipo INT.

**Nombre:** aplicando el mismo criterio que con los títulos, usaremos el tipo VARCHAR(60).

### **Relación Editorial**

**ClaveEditorial:** no existen demasiadas editoriales, probablemente un SMALLINT sea suficiente.

**Nombre:** usaremos el mismo criterio que para títulos y nombres de autores, VARCHAR(60).

**Dirección:** también VARCHAR(60).

**Teléfono:** los números de teléfono, a pesar de ser números, no se suelen almacenar como tales. El problema es que a veces se incluyen otros caracteres, como el '+' para el prefijo, o los paréntesis. En ciertos países se usan caracteres como sinónimos de dígitos, etc. Usaremos una cadena lo bastante larga, VARCHAR(15).

### **Relación Ejemplar**

**ClaveLibro:** es una clave foránea, el tipo debe ser INT.

**NúmeroOrden:** tal vez el tipo TINYINT sea pequeño en algunos casos, de modo que usaremos SMALLINT.

**Edición:** pasa lo mismo que con el valor anterior, puede haber libros con más de 255 ediciones, no podemos arriesgarnos. Usaremos SMALLINT.

**Ubicación:** esto depende de cómo se organice la biblioteca, pero un campo de texto puede almacenar tanto coordenadas como etiquetas, podemos usar un VARCHAR(15).

## **Relación Socio**

**ClaveSocio:** usaremos un INT.

**Nombre:** seguimos con el mismo criterio, VARCHAR(60).

**Dirección:** lo mismo, VARCHAR(60).

**Teléfono:** como en el caso de la editorial, VARCHAR(15).

**Categoría:** ya vimos que este atributo es un carácter, CHAR.

## **Relación Préstamo**

**ClaveSocio:** como clave foránea, el tipo está predeterminado. INT.

**ClaveLibro:** igual que el anterior, INT.

**NúmeroOrden:** y lo mismo en este caso, SMALLINT.

**Fecha\_préstamo:** tipo DATE, sólo almacenamos la fecha.

**Fecha\_devolución:** también DATE.

**Notas:** necesitamos espacio, para este tipo de atributos se usa el tipo BLOB.

## **Relación Trata\_sobre**

**ClaveLibro:** INT.

**ClaveTema:** SMALLINT.

**Relación** *Escrito\_por*

**ClaveLibro:** INT.

**ClaveAutor:** INT.

## 6 El cliente MySQL

Bien, ha llegado el momento de empezar a trabajar con el SGBD, es decir, vamos a empezar a entendernos con **MySQL**.

Existen muchas formas de establecer una comunicación con el servidor de **MySQL**. En nuestros programas, generalmente, usaremos un API para realizar las consultas con el servidor. En PHP, por ejemplo, este API está integrado con el lenguaje, en C/C++ se trata de librerías de enlace dinámico, etc.

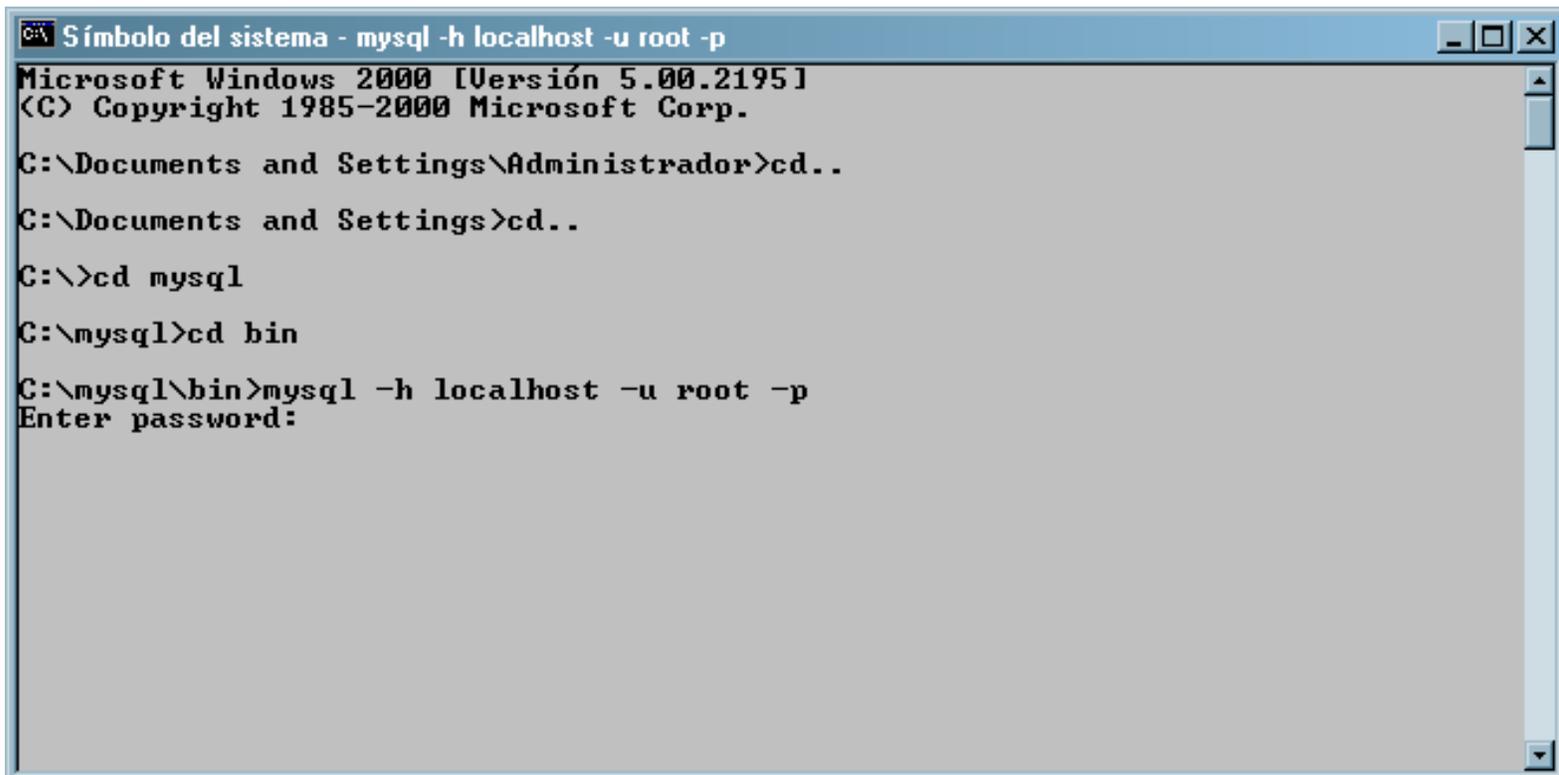
Para este curso usaremos **MySQL** de forma directa, mediante un cliente ejecutándose en una consola (una ventana DOS en Windows, o un Shell en otros sistemas). En otras secciones se explicarán los diferentes APIs.

Veamos un ejemplo sencillo. Para ello abrimos una consola y tecleamos "mysql". (Si estamos en Windows y no está definido el camino para **MySQL** tendremos que hacerlo desde "C:\mysql\bin").

Para entrar en la consola de **MySQL** se requieren ciertos parámetros. Hay que tener en cuenta que el servidor es multiusuario, y que cada usuario puede tener distintos privilegios, tanto de acceso a tablas como de comandos que puede utilizar.

La forma general de iniciar una sesión **MySQL** es:

```
mysql -h host -u usuario -p
```



```
Símbolo del sistema - mysql -h localhost -u root -p
Microsoft Windows 2000 [Versión 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.
C:\Documents and Settings\Administrador>cd..
C:\Documents and Settings>cd..
C:\>cd mysql
C:\mysql>cd bin
C:\mysql\bin>mysql -h localhost -u root -p
Enter password:
```

Podemos especificar el ordenador donde está el servidor de bases de datos (host) y nuestro nombre de usuario. Los

parámetros "-h" y "-u" indican que los parámetros a continuación son, respectivamente, el nombre del host y el usuario. El parámetro "-p" indica que se debe solicitar una clave de acceso.

En versiones de **MySQL** anteriores a la 4.1.9 es posible abrir un cliente de forma anónima sin especificar una contraseña. Pero esto es mala idea, y de hecho, las últimas versiones de **MySQL** no lo permiten. Durante la instalación de **MySQL** se nos pedirá que elijamos una clave de acceso para el usuario 'root', deberemos usar esa clave para iniciar una sesión con el cliente **MySQL**.

```
mysql -h localhost -u root -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 76 to server version: 4.1.9-nt

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

mysql>
```

Para salir de una sesión del cliente de **MySQL** se usa el comando "QUIT".

```
mysql> QUIT
Bye

C:\mysql\bin>
```

## Algunas consultas

Ahora ya sabemos entrar y salir del cliente **MySQL**, y podemos hacer consultas. Lo más sencillo es consultar algunas variables del sistema o el valor de algunas funciones de **MySQL**. Para hacer este tipo de consultas se usa la sentencia SQL **SELECT**, por ejemplo:

```
mysql> SELECT VERSION(), CURRENT_DATE;
+-----+-----+
| VERSION() | CURRENT_DATE |
+-----+-----+
| 4.0.15-nt | 2003-12-12   |
+-----+-----+
1 row in set (0.02 sec)

mysql>
```

**SELECT** es la sentencia SQL para seleccionar datos de bases de datos, pero también se puede usar, como en este caso, para consultar variables del sistema o resultados de funciones. En este caso hemos consultado el resultado de la función **VERSION** y de la variable **CURRENT\_DATE**.

Esto es sólo un aperitivo, hay muchísimas más opciones, y mucho más interesantes.

En los próximos capítulos, antes de entrar en profundidad en esta sentencia, usaremos **SELECT** para mostrar todas las filas de una tabla. Para ello se usa la sentencia:

```
mysql> SELECT * FROM <tabla>;
```

## Usuarios y privilegios

Cuando trabajemos con bases de datos reales y con aplicaciones de gestión de bases de datos, será muy importante definir otros usuarios, además del root, que es el administrador. Esto nos permitirá asignar distintos privilegios a cada usuario, y nos ayudará a proteger las bases de datos.

Podremos por ejemplo, crear un usuario que sólo tenga posibilidad de consultar datos de determinadas tablas o bases de datos, pero que no tenga permiso para añadir o modificar datos, o modificar la estructura de la base de datos.

Otros usuarios podrán insertar datos, y sólo algunos (o mejor, sólo uno) podrán modificar la estructura de las bases de datos: los administradores.

Dedicaremos un capítulo completo a la gestión de usuarios y privilegios, pero de momento trabajaremos siempre con todos ellos, de modo que tendremos cuidado con lo que hacemos. Además, trabajaremos sólo con las bases de datos de ejemplo.

# 7 Lenguaje SQL

## Creación de bases de datos y tablas

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

DDL (Data Definition Language) Lenguaje de definición de datos. Es el lenguaje que se usa para crear bases de datos y tablas, y para modificar sus estructuras, así como los permisos y privilegios.

Este lenguaje trabaja sobre unas tablas especiales llamadas *diccionario de datos*.

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

SQL engloba ambos lenguajes DDL+DML, y los estudiaremos juntos, ya que ambos forman parte del conjunto de sentencias de SQL.

En este capítulo vamos a explicar el proceso para pasar del modelo lógico relacional, en forma de esquemas de relaciones, al modelo físico, usando sentencias SQL, y viendo las peculiaridades específicas de **MySQL**.

### Crear una base de datos

Cada conjunto de relaciones que componen un modelo completo forma una base de datos. Desde el punto de vista de SQL, una base de datos es sólo un conjunto de relaciones (o tablas), y para organizarlas o distinguirlas se accede a ellas mediante su nombre. A nivel de sistema operativo, cada base de datos se guarda en un directorio diferente.

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

Vamos a crear y manipular nuestra propia base de datos, al tiempo que nos familiarizamos con la forma de trabajar de **MySQL**.

Para empezar, crearemos una base de datos para nosotros solos, y la llamaremos "prueba". Para crear una base de datos se usa una sentencia **CREATE DATABASE**:

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

```
mysql>
```

Podemos averiguar cuántas bases de datos existen en nuestro sistema usando la sentencia [SHOW DATABASES](#):

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

mysql>
```

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

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

## Crear una tabla

Veamos ahora la sentencia [CREATE TABLE](#) que sirve para crear tablas.

La sintaxis de esta sentencia es muy compleja, ya que existen muchas opciones y tenemos muchas posibilidades diferentes a la hora de crear una tabla. Las iremos viendo paso a paso, y en poco tiempo sabremos usar muchas de sus posibilidades.

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

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

mysql>
```

Hemos creado una tabla llamada "gente" con dos columnas: "nombre" que puede contener cadenas de hasta 40 caracteres y "fecha" de tipo fecha.

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

```
mysql> SHOW TABLES;
+-----+
| Tables_in_prueba |
+-----+
| gente             |
+-----+
1 row in set (0.01 sec)

mysql>
```

Pero tenemos muchas más opciones a la hora de definir columnas. Además del tipo y el nombre, podemos definir valores por defecto, permitir o no que contengan valores nulos, crear una clave primaria, indexar...

La sintaxis para definir columnas es:

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

Veamos cada una de las opciones por separado.

## Valores nulos

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

Debemos recordar que, como vimos en los capítulos de modelado, aquellas columnas que son o forman parte de una clave primaria no pueden contener valores nulos.

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

La opción por defecto es que se permitan valores nulos, *NULL*, y para que no se permitan, se usa *NOT NULL*. Por ejemplo:

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

## Valores por defecto

Para cada columna también se puede definir, opcionalmente, un valor por defecto. El valor por defecto se asignará de forma automática a una columna cuando no se especifique un valor determinado al añadir filas.

Si una columna puede tener un valor nulo, y no se especifica un valor por defecto, se usará *NULL* como valor por defecto. En el ejemplo anterior, el valor por defecto para *poblacion* es *NULL*.

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

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

## Claves primarias

También se puede definir una clave primaria sobre una columna, usando la palabra clave *KEY* o *PRIMARY KEY*.

Sólo puede existir una clave primaria en cada tabla, y la columna sobre la que se define una clave primaria no puede tener valores *NULL*. Si esto no se especifica de forma explícita, **MySQL** lo hará de forma automática.

Por ejemplo, si queremos crear un índice en la columna *nombre* de la tabla de ciudades, crearemos la tabla así:

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

Usar *NOT NULL PRIMARY KEY* equivale a *PRIMARY KEY, NOT NULL KEY* o sencillamente *KEY*. Personalmente, prefiero usar la primera forma o la segunda.

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

## Columnas autoincrementadas

En **MySQL** tenemos la posibilidad de crear una columna autoincrementada, aunque esta columna sólo puede ser de tipo entero.

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

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

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

## Comentarios

Adicionalmente, al crear la tabla, podemos añadir un comentario a cada columna. Este comentario

sirve como información adicional sobre alguna característica especial de la columna, y entra en el apartado de documentación de la base de datos:

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

## Definición de creación

A continuación de las definiciones de las columnas podemos añadir otras definiciones. La sintaxis más general es:

```
definición_columnas
| [CONSTRAINT [símbolo]] PRIMARY KEY (index_nombre_col,...)
| KEY [nombre_index] (nombre_col_index,...)
| INDEX [nombre_index] (nombre_col_index,...)
| [CONSTRAINT [símbolo]] UNIQUE [INDEX]
|     [nombre_index] [tipo_index] (nombre_col_index,...)
| [FULLTEXT|SPATIAL] [INDEX] [nombre_index] (nombre_col_index,...)
| [CONSTRAINT [símbolo]] FOREIGN KEY
|     [nombre_index] (nombre_col_index,...) [definición_referencia]
| CHECK (expr)
```

Veremos ahora cada una de estas opciones.

## Índices

Tenemos tres tipos de índices. El primero corresponde a las claves primarias, que como vimos, también se pueden crear en la parte de definición de columnas.

### Claves primarias

La sintaxis para definir claves primarias es:

```
definición_columnas
| PRIMARY KEY (index_nombre_col,...)
```

El ejemplo anterior que vimos para crear claves primarias, usando esta sintaxis, quedaría así:

```
mysql> CREATE TABLE ciudad4 (nombre CHAR(20) NOT NULL,  
-> poblacion INT NULL DEFAULT 5000,  
-> PRIMARY KEY (nombre));  
Query OK, 0 rows affected (0.17 sec)
```

Pero esta forma tiene más opciones, por ejemplo, entre los paréntesis podemos especificar varios nombres de columnas, para construir claves primarias compuestas por varias columnas:

```
mysql> CREATE TABLE mitabla1 (  
-> id1 CHAR(2) NOT NULL,  
-> id2 CHAR(2) NOT NULL,  
-> texto CHAR(30),  
-> PRIMARY KEY (id1, id2));  
Query OK, 0 rows affected (0.09 sec)
```

```
mysql>
```

## Índices

El segundo tipo de índice permite definir índices sobre una columna, sobre varias, o sobre partes de columnas. Para definir estos índices se usan indistintamente las opciones *KEY* o *INDEX*.

```
mysql> CREATE TABLE mitabla2 (  
-> id INT,  
-> nombre CHAR(19),  
-> INDEX (nombre));  
Query OK, 0 rows affected (0.09 sec)
```

O su equivalente:

```
mysql> CREATE TABLE mitabla3 (  
-> id INT,  
-> nombre CHAR(19),  
-> KEY (nombre));  
Query OK, 0 rows affected (0.09 sec)
```

También podemos crear un índice sobre parte de una columna:

```
mysql> CREATE TABLE mitabla4 (
-> id INT,
-> nombre CHAR(19),
-> INDEX (nombre(4)));
Query OK, 0 rows affected (0.09 sec)
```

Este ejemplo usará sólo los cuatro primeros caracteres de la columna 'nombre' para crear el índice.

## Claves únicas

El tercero permite definir índices con claves únicas, también sobre una columna, sobre varias o sobre partes de columnas. Para definir índices con claves únicas se usa la opción *UNIQUE*.

La diferencia entre un índice único y uno normal es que en los únicos no se permite la inserción de filas con claves repetidas. La excepción es el valor *NULL*, que sí se puede repetir.

```
mysql> CREATE TABLE mitabla5 (
-> id INT,
-> nombre CHAR(19),
-> UNIQUE (nombre));
Query OK, 0 rows affected (0.09 sec)
```

Una clave primaria equivale a un índice de clave única, en la que el valor de la clave no puede tomar valores *NULL*. Tanto los índices normales como los de claves únicas sí pueden tomar valores *NULL*.

Por lo tanto, las definiciones siguientes son equivalentes:

```
mysql> CREATE TABLE mitabla6 (
-> id INT,
-> nombre CHAR(19) NOT NULL,
-> UNIQUE (nombre));
Query OK, 0 rows affected (0.09 sec)
```

Y:

```
mysql> CREATE TABLE mitabla7 (
```

```
-> id INT,  
-> nombre CHAR(19),  
-> PRIMARY KEY (nombre));  
Query OK, 0 rows affected (0.09 sec)
```

Los índices sirven para optimizar las consultas y las búsquedas de datos. Mediante su uso es mucho más rápido localizar filas con determinados valores de columnas, o seguir un determinado orden. La alternativa es hacer búsquedas secuenciales, que en tablas grandes requieren mucho tiempo.

## Claves foráneas

En **MySQL** sólo existe soporte para claves foráneas en tablas de tipo **InnoDB**. Sin embargo, esto no impide usarlas en otros tipos de tablas.

La diferencia consiste en que en esas tablas no se verifica si una clave foránea existe realmente en la tabla referenciada, y que no se eliminan filas de una tabla con una definición de clave foránea. Para hacer esto hay que usar tablas **InnoDB**.

Hay dos modos de definir claves foráneas en bases de datos **MySQL**.

El primero, sólo sirve para documentar, y, al menos en las pruebas que he hecho, no define realmente claves foráneas. Esta forma consiste en definir una referencia al mismo tiempo que se define una columna:

```
mysql> CREATE TABLE personas (  
-> id INT AUTO_INCREMENT PRIMARY KEY,  
-> nombre VARCHAR(40),  
-> fecha DATE);  
Query OK, 0 rows affected (0.13 sec)  
  
mysql> CREATE TABLE telefonos (  
-> numero CHAR(12),  
-> id INT NOT NULL REFERENCES personas (id)  
-> ON DELETE CASCADE ON UPDATE CASCADE); (1)  
Query OK, 0 rows affected (0.13 sec)  
  
mysql>
```

Hemos usado una definición de referencia para la columna 'id' de la tabla 'telefonos', indicando que es una clave foránea correspondiente a la columna 'id' de la tabla 'personas' (1). Sin embargo, aunque la sintaxis se comprueba, esta definición no implica ningún comportamiento por parte de **MySQL**.

La otra forma es mucho más útil, aunque sólo se aplica a tablas **InnoDB**.

En esta forma no se añade la referencia en la definición de la columna, sino después de la definición de todas las columnas. Tenemos la siguiente sintaxis resumida:

```
CREATE TABLE nombre
  definición_de_columnas
  [CONSTRAINT [símbolo]] FOREIGN KEY [nombre_index]
(nombre_col_index,...)
  [REFERENCES nombre_tabla [(nombre_col,...)]
  [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
  [ON DELETE RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT]
  [ON UPDATE RESTRICT | CASCADE | SET NULL | NO ACTION | SET
DEFAULT]]
```

El ejemplo anterior, usando tablas **InnoDB** y esta definición de claves foráneas quedará así:

```
mysql> CREATE TABLE personas2 (
-> id INT AUTO_INCREMENT PRIMARY KEY,
-> nombre VARCHAR(40),
-> fecha DATE)
-> ENGINE=InnoDB;
Query OK, 0 rows affected (0.13 sec)

mysql> CREATE TABLE telefonos2 (
-> numero CHAR(12),
-> id INT NOT NULL,
-> KEY (id), (2)
-> FOREIGN KEY (id) REFERENCES personas2 (id)
-> ON DELETE CASCADE ON UPDATE CASCADE) (3)
-> ENGINE=InnoDB;
Query OK, 0 rows affected (0.13 sec)

mysql>
```

Es imprescindible que la columna que contiene una definición de clave foránea esté indexada (2). Pero esto no debe preocuparnos demasiado, ya que si no lo hacemos de forma explícita, **MySQL** lo hará por nosotros de forma implícita.

Esta forma define una clave foránea en la columna 'id', que hace referencia a la columna 'id' de la tabla 'personas' (3). La definición incluye las tareas a realizar en el caso de que se elimine una fila en la tabla 'personas'.

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

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

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

- **RESTRICT:** esta opción impide eliminar o modificar filas en la tabla referenciada si existen filas con el mismo valor de clave foránea.
- **CASCADE:** borrar o modificar una clave en una fila en la tabla referenciada con un valor determinado de clave, implica borrar las filas con el mismo valor de clave foránea o modificar los valores de esas claves foráneas.
- **SET NULL:** borrar o modificar una clave en una fila en la tabla referenciada con un valor determinado de clave, implica asignar el valor *NULL* a las claves foráneas con el mismo valor.
- **NO ACTION:** las claves foráneas no se modifican, ni se eliminan filas en la tabla que las contiene.
- **SET DEFAULT:** borrar o modificar una clave en una fila en la tabla referenciada con un valor determinado implica asignar el valor por defecto a las claves foráneas con el mismo valor.

Por ejemplo, veamos esta definición de la tabla 'telefonos':

```
mysql> CREATE TABLE telefonos3 (
-> numero CHAR(12),
-> id INT NOT NULL,
-> KEY (id),
-> FOREIGN KEY (id) REFERENCES personas (id)
-> ON DELETE RESTRICT ON UPDATE CASCADE)
-> ENGINE=InnoDB;
```

Query OK, 0 rows affected (0.13 sec)

```
mysql>
```

Si se intenta borrar una fila de 'personas' con un determinado valor de 'id', se producirá un error si existen filas en la tabla 'telefonos3' con mismo valor en la columna 'id'. La fila de 'personas' no se eliminará, a no ser que previamente eliminemos las filas con el mismo valor de clave foránea en 'teléfonos3'.

Si se modifica el valor de la columna 'id' en la tabla 'personas', se modificarán los valores de la columna 'id' para mantener la relación.

Veamos un ejemplo más práctico:

**personas**

<u>id</u>	<u>nombre</u>	<u>fecha</u>
1	Fulanito	1998/04/14
2	Menganito	1975/06/18
3	Tulanito	1984/07/05

**telefonos3**

<u>numero</u>	<u>id</u>
12322132	1
12332221	1
55546545	3
55565445	3

Si intentamos borrar la fila correspondiente a "Fulanito" se producirá un error, ya que existen dos filas en 'telefonos' con el valor 1 en la columna 'id'.

Sí será posible borrar la fila correspondiente a "Menganito", ya que no existe ninguna fila en la tabla 'telefonos' con el valor 2 en la columna 'id'.

Si modificamos el valor de 'id' en la fila correspondiente a "Tulanito", por el valor 4, por ejemplo, se asignará el valor 4 a la columna 'id' de las filas 3ª y 4ª de la tabla 'telefonos3':

**personas**

<u>id</u>	<u>nombre</u>	<u>fecha</u>
1	Fulanito	1998/04/14
2	Menganito	1975/06/18
<b>4</b>	Tulanito	1984/07/05

**telefonos3**

<u>numero</u>	<u>id</u>
12322132	1
12332221	1
55546545	<b>4</b>
55565445	<b>4</b>

No hemos usado todas las opciones. Las opciones de MATCH FULL, MATCH PARTIAL o MATCH SIMPLE no las comentaremos de momento (lo dejaremos para más adelante).

La parte opcional CONSTRAINT [símbolo] sirve para asignar un nombre a la clave foránea, de modo que pueda usarse como identificador si se quiere modificar o eliminar una definición de clave foránea.

También veremos esto con más detalle en capítulos avanzados.

## Otras opciones

Las opciones como *FULLTEXT* o *SPATIAL* las veremos en otras secciones.

La opción *CHECK* no está implementada en **MySQL**.

## Opciones de tabla

La parte final de la sentencia [CREATE TABLE](#) permite especificar varias opciones para la tabla.

Sólo comentaremos la opción del motor de almacenamiento, para ver el resto en detalle se puede consultar la sintaxis en [CREATE TABLE](#).

## Motor de almacenamiento

La sintaxis de esta opción es:

```
{ENGINE | TYPE} = {BDB | HEAP | ISAM | InnoDB | MERGE | MRG_MYISAM | MYISAM }
```

Podemos usar indistintamente *ENGINE* o *TYPE*, pero la forma recomendada es *ENGINE* ya que la otra desaparecerá en la versión 5 de **MySQL**.

Hay seis motores de almacenamiento disponibles. Algunos de ellos serán de uso obligatorio si queremos tener ciertas opciones disponibles. Por ejemplo, ya hemos comentado que el soporte para claves foráneas sólo está disponible para el motor **InnoDB**. Los motores son:

- **BerkeleyDB o BDB:** tablas de transacción segura con bloqueo de página.
- **HEAP o MEMORY:** tablas almacenadas en memoria.
- **ISAM:** motor original de **MySQL**.
- **InnoDB:** tablas de transacción segura con bloqueo de fila y claves foráneas.
- **MERGE o MRG\_MyISAM:** una colección de tablas MyISAM usadas como una única tabla.
- **MyISAM:** el nuevo motor binario de almacenamiento portable que reemplaza a ISAM.

Generalmente usaremos tablas **MyISAM** o tablas **InnoDB**.

A veces, cuando se requiera una gran optimización, creamos tablas temporales en memoria.

## Verificaciones

Disponemos de varias sentencias para verificar o consultar características de tablas.

Podemos ver la estructura de una tabla usando la sentencia [SHOW COLUMNS](#):

```
mysql> SHOW COLUMNS FROM gente;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| nombre | varchar(40)   | YES  |     | NULL    |      |
| fecha  | date          | YES  |     | NULL    |      |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

También podemos ver la instrucción usada para crear una tabla, mediante la sentencia [SHOW CREATE TABLE](#):

```
mysql> show create table gente\G
***** 1. row *****
      Table: gente
Create Table: CREATE TABLE `gente` (
  `nombre` varchar(40) default NULL,
  `fecha` date default NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)

mysql>
```

Usando '\G' en lugar de ';' la salida se muestra en forma de listado, en lugar de en forma de tabla. Este formato es más cómodo a la hora de leer el resultado.

La sentencia [CREATE TABLE](#) mostrada no tiene por qué ser la misma que se usó al crear la tabla originalmente. **MySQL** rellena las opciones que se activan de forma implícita, y usa siempre el mismo formato para crear claves primarias.

## Eliminar una tabla

A veces es necesario eliminar una tabla, ya sea porque es más sencillo crearla de nuevo que modificarla, o porque ya no es necesaria.

Para eliminar una tabla se usa la sentencia [DROP TABLE](#).

La sintaxis es simple:

```
DROP TABLE [IF EXISTS] tbl_name [, tbl_name] ...
```

Por ejemplo:

```
mysql> DROP TABLE ciudad6;  
Query OK, 0 rows affected (0.75 sec)  
  
mysql>
```

Se pueden añadir las palabras *IF EXISTS* para evitar errores si la tabla a eliminar no existe.

```
mysql> DROP TABLE ciudad6;  
ERROR 1051 (42S02): Unknown table 'ciudad6'  
mysql> DROP TABLE IF EXISTS ciudad6;  
Query OK, 0 rows affected, 1 warning (0.00 sec)  
  
mysql>
```

## Eliminar una base de datos

De modo parecido, se pueden eliminar bases de datos completas, usando la sentencia [DROP DATABASE](#).

La sintaxis también es muy simple:

```
DROP DATABASE [IF EXISTS] db_name
```

Hay que tener cuidado, ya que al borrar cualquier base de datos se elimina también cualquier tabla que contenga.

```
mysql> CREATE DATABASE borrame;
Query OK, 1 row affected (0.00 sec)

mysql> USE borrame
Database changed
mysql> CREATE TABLE borrame (
  -> id INT,
  -> nombre CHAR(40)
  -> );
Query OK, 0 rows affected (0.09 sec)

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

mysql> SHOW TABLES;
+-----+
| Tables_in_borrame |
+-----+
| borrame            |
+-----+
1 row in set (0.00 sec)

mysql> DROP DATABASE IF EXISTS borrame;
Query OK, 1 row affected (0.11 sec)

mysql> DROP DATABASE IF EXISTS borrame;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql>
```

## Ejemplo 1

Veamos ahora como crear las bases de datos y tablas correspondientes a los ejemplos que vamos siguiendo desde el principio del curso.

Nuestro primer ejemplo tenía este esquema:

```

Estación(Identificador, Latitud, Longitud, Altitud)
Muestra(IdentificadorEstacion, Fecha, Temperatura mínima, Temperatura
máxima,
    Precipitaciones, Humedad mínima, Humedad máxima, Velocidad del viento
mínima,
    Velocidad del viento máxima)

```

Y habíamos elegido los siguientes tipos para cada columna:

Columna	Tipo
Identificador	MEDIUMINT UNSIGNED
Latitud	VARCHAR(14)
Longitud	VARCHAR(15)
Altitud	MEDIUMINT
IdentificadorEstacion	MEDIUMINT UNSIGNED
Fecha	DATE
Temperatura mínima	TINYINT
Temperatura máxima	TINYINT
Precipitaciones	SMALLINT UNSIGNED
Humedad mínima	TINYINT UNSIGNED
Humedad máxima	TINYINT UNSIGNED
Velocidad del viento mínima	SMALLINT UNSIGNED
Velocidad del viento máxima	SMALLINT UNSIGNED

Primero crearemos una base de datos, a la que podemos llamar "meteo":

```

mysql> CREATE DATABASE meteo;
Query OK, 1 row affected (0.02 sec)

```

Podemos situarnos en la base de datos, usando la [USE](#) o bien crear las tablas usando el nombre completo, es decir, mediante el prefijo "meteo.". Empecemos con la primera tabla:

```

mysql> USE meteo
Database changed

```

```
mysql> CREATE TABLE estacion (
-> identificador MEDIUMINT UNSIGNED NOT NULL,
-> latitud VARCHAR(14) NOT NULL,
-> longitud VARCHAR(15) NOT NULL,
-> altitud MEDIUMINT NOT NULL,
-> PRIMARY KEY (identificador)
-> ) ENGINE=InnoDB;
Query OK, 0 rows affected (0.30 sec)
```

Hemos creado una clave primaria, y como es obligatorio, hemos definido la columna 'identificador' como *NOT NULL*. Usaremos tablas **InnoDB** ya que queremos que **MySQL** haga el control de las claves foráneas.

Veamos ahora la segunda tabla:

```
mysql> CREATE TABLE muestra (
-> identifiadorestacion MEDIUMINT UNSIGNED NOT NULL,
-> fecha DATE NOT NULL,
-> temperaturaminima TINYINT,
-> temperaturamaxima TINYINT,
-> precipitaciones SMALLINT UNSIGNED,
-> humedadminima TINYINT UNSIGNED,
-> humedadamaxima TINYINT UNSIGNED,
-> velocidadminima SMALLINT UNSIGNED,
-> velocidadmaxima SMALLINT UNSIGNED,
-> KEY (identifiadorestacion),
-> FOREIGN KEY (identifiadorestacion)
-> REFERENCES estacion(identificador)
-> ON DELETE NO ACTION
-> ON UPDATE CASCADE
-> ) ENGINE=InnoDB;
Query OK, 0 rows affected (0.16 sec)
```

A falta de datos concretos en el enunciado sobre qué hacer con las muestras de estaciones si estas desaparecen, hemos optado por mantener el valor de la clave foránea. También hemos decidido modificar el identificador si se cambia en la tabla de estaciones.

## Ejemplo 2

El segundo ejemplo consiste en modelar una biblioteca. Este era el esquema:

```

Libro(ClaveLibro, Título, Idioma, Formato, Categoría, ClaveEditorial)
Tema(ClaveTema, Nombre)
Autor(ClaveAutor, Nombre)
Editorial(ClaveEditorial, Nombre, Dirección, Teléfono)
Ejemplar(ClaveLibro, NúmeroOrden, Edición, Ubicación)
Socio(ClaveSocio, Nombre, Dirección, Teléfono, Categoría)
Préstamo(ClaveSocio, ClaveLibro, NúmeroOrden, Fecha_préstamo,
    Fecha_devolución, Notas)
Trata_sobre(ClaveLibro, ClaveTema)
Escrito_por(ClaveLibro, ClaveAutor)

```

Y los tipos para las columnas:

Columna	Tipo
ClaveLibro	INT
Titulo	VARCHAR(60)
Idioma	VARCHAR(15)
Formato	VARCHAR(15)
Categoria(libro)	CHAR
ClaveTema	SMALLINT
Nombre(tema)	VARCHAR(40)
ClaveAutor	INT
Nombre(autor)	VARCHAR(60)
ClaveEditorial	SMALLINT
Nombre(editorial)	VARCHAR(60)
Direccion(editorial)	VARCHAR(60)
Telefono(editorial)	VARCHAR(15)
NumeroOrden	SMALLINT
Edicion	SMALLINT
Ubicacion	VARCHAR(15)
ClaveSocio	INT
Nombre(socio)	VARCHAR(60)
Direccion(socio)	VARCHAR(60)
Telefono(socio)	VARCHAR(15)
Categoria(socio)	CHAR

Fecha_prestamo	DATE
Fecha_devolucion	DATE
Notas	BLOB

Empecemos por crear la base de datos:

```
mysql> CREATE DATABASE biblio;
Query OK, 1 row affected (0.16 sec)

mysql>USE biblio
Database changed
mysql>
```

Ahora crearemos las tablas. Primero crearemos la tabla de editoriales, ya que su clave primaria se usa como clave foránea en la tabla de libros:

```
mysql> CREATE TABLE editorial (
-> claveeditorial SMALLINT NOT NULL,
-> nombre VARCHAR(60),
-> direccion VARCHAR(60),
-> telefono VARCHAR(15),
-> PRIMARY KEY (claveeditorial)
-> ) ENGINE=InnoDB;
Query OK, 0 rows affected (0.09 sec)

mysql>
```

Ahora podemos crear la tabla de libros:

```
mysql> CREATE TABLE libro (
-> clavelibro INT NOT NULL,
-> titulo VARCHAR(60),
-> idioma VARCHAR(15),
-> formato VARCHAR(15),
-> categoria CHAR,
-> claveeditorial SMALLINT,
-> PRIMARY KEY (clavelibro),
-> KEY(claveeditorial),
-> FOREIGN KEY (claveeditorial)
-> REFERENCES editorial(claveeditorial)
```

```
-> ON DELETE SET NULL
-> ON UPDATE CASCADE
-> ) ENGINE=InnoDB;
Query OK, 0 rows affected (0.11 sec)

mysql>
```

Seguimos con las tablas de tema, autor:

```
mysql> CREATE TABLE tema (
-> clavetema SMALLINT NOT NULL,
-> nombre VARCHAR(40),
-> PRIMARY KEY (clavetema)
-> ) ENGINE=InnoDB;
Query OK, 0 rows affected (0.06 sec)

mysql> CREATE TABLE autor (
-> claveautor INT NOT NULL,
-> nombre VARCHAR(60),
-> PRIMARY KEY (claveautor)
-> ) ENGINE=InnoDB;
Query OK, 0 rows affected (0.09 sec)

mysql>
```

Seguimos con la tabla ejemplar, que recordemos que contiene entidades subordinadas de libro:

```
mysql> CREATE TABLE ejemplar (
-> clavelibro INT NOT NULL,
-> numeroorden SMALLINT NOT NULL,
-> edicion SMALLINT,
-> ubicacion VARCHAR(15),
-> PRIMARY KEY (clavelibro,numeroorden),
-> FOREIGN KEY (clavelibro)
-> REFERENCES libro(clavelibro)
-> ON DELETE CASCADE
-> ON UPDATE CASCADE
-> ) ENGINE=InnoDB;
Query OK, 0 rows affected (0.11 sec)

mysql>
```

Seguimos con la tabla de socio:

```
mysql> CREATE TABLE socio (  
-> clavesocio INT NOT NULL,  
-> nombre VARCHAR(60),  
-> direccion VARCHAR(60),  
-> telefono VARCHAR(15),  
-> categoria CHAR,  
-> PRIMARY KEY (clavesocio)  
-> ) ENGINE=InnoDB;  
Query OK, 0 rows affected (0.08 sec)  
  
mysql>
```

Ahora la tabla de prestamo:

```
mysql> CREATE TABLE prestamo (  
-> clavesocio INT,  
-> clavelibro INT,  
-> numeroorden SMALLINT,  
-> fecha_prestamo DATE NOT NULL,  
-> fecha_devolucion DATE DEFAULT NULL,  
-> notas BLOB,  
-> FOREIGN KEY (clavesocio)  
-> REFERENCES socio(clavesocio)  
-> ON DELETE SET NULL  
-> ON UPDATE CASCADE,  
-> FOREIGN KEY (clavelibro)  
-> REFERENCES ejemplar(clavelibro)  
-> ON DELETE SET NULL  
-> ON UPDATE CASCADE  
-> ) ENGINE=InnoDB;  
Query OK, 0 rows affected (0.16 sec)  
  
mysql>
```

Sólo nos quedan por crear las tablas trata\_sobre y escrito\_por:

```
mysql> CREATE TABLE trata_sobre (  
-> clavelibro INT NOT NULL,  
-> clavetema SMALLINT NOT NULL,  
-> FOREIGN KEY (clavelibro)
```

```
-> REFERENCES libro(clavelibro)
-> ON DELETE CASCADE
-> ON UPDATE CASCADE,
-> FOREIGN KEY (clavetema)
-> REFERENCES tema(clavetema)
-> ON DELETE CASCADE
-> ON UPDATE CASCADE
-> ) ENGINE=InnoDB;
```

Query OK, 0 rows affected (0.11 sec)

```
mysql> CREATE TABLE escrito_por (
-> clavelibro INT NOT NULL,
-> claveautor INT NOT NULL,
-> FOREIGN KEY (clavelibro)
-> REFERENCES libro(clavelibro)
-> ON DELETE CASCADE
-> ON UPDATE CASCADE,
-> FOREIGN KEY (claveautor)
-> REFERENCES autor(claveautor)
-> ON DELETE CASCADE
-> ON UPDATE CASCADE
-> ) ENGINE=InnoDB;
```

Query OK, 0 rows affected (0.13 sec)

mysql>

## 8 Lenguaje SQL

### Inserción y modificación de datos

Llegamos ahora a un punto interesante, una base de datos sin datos no sirve para mucho, de modo que veremos cómo agregar, modificar o eliminar los datos que contienen nuestras bases de datos.

#### Inserción de nuevas filas

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

```
mysql> INSERT INTO gente VALUES ('Fulano','1974-04-12');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO gente VALUES ('Mengano','1978-06-15');
Query OK, 1 row affected (0.04 sec)

mysql> INSERT INTO gente VALUES
-> ('Tulano','2000-12-02'),
-> ('Pegano','1993-02-10');
Query OK, 2 rows affected (0.02 sec)
Records: 2 Duplicates: 0 Warnings: 0

mysql> SELECT * FROM gente;
+-----+-----+
| nombre | fecha      |
+-----+-----+
| Fulano  | 1974-04-12 |
| Mengano | 1978-06-15 |
| Tulano  | 2000-12-02 |
| Pegano  | 1993-02-10 |
+-----+-----+
4 rows in set (0.08 sec)

mysql>
```

Si no necesitamos asignar un valor concreto para alguna columna, podemos asignarle el valor por defecto indicado para esa columna cuando se creó la tabla, usando la palabra *DEFAULT*:

```
mysql> INSERT INTO ciudad2 VALUES ('Perillo', DEFAULT);
Query OK, 1 row affected (0.03 sec)

mysql> SELECT * FROM ciudad2;
+-----+-----+
| nombre | poblacion |
+-----+-----+
| Perillo |      5000 |
+-----+-----+
1 row in set (0.02 sec)
mysql>
```

En este caso, como habíamos definido un valor por defecto para población de 5000, se asignará ese valor para la fila correspondiente a 'Perillo'.

Otra opción consiste en indicar una lista de columnas para las que se van a suministrar valores. A las columnas que no se nombren en esa lista se les asigna el valor por defecto. Este sistema, además, permite usar cualquier orden en las columnas, con la ventaja, con respecto a la anterior forma, de que no necesitamos conocer el orden de las columnas en la tabla para poder insertar datos:

```
mysql> INSERT INTO ciudad5 (poblacion,nombre) VALUES
-> (7000000, 'Madrid'),
-> (9000000, 'París'),
-> (3500000, 'Berlín');
Query OK, 3 rows affected (0.05 sec)
Records: 3 Duplicates: 0 Warnings: 0

mysql> SELECT * FROM ciudad5;
+-----+-----+-----+
| clave | nombre | poblacion |
+-----+-----+-----+
|      1 | Madrid |  7000000 |
|      2 | París  |  9000000 |
|      3 | Berlín |  3500000 |
+-----+-----+-----+
3 rows in set (0.03 sec)

mysql>
```

Cuando creamos la tabla "ciudad5" definimos tres columnas: 'clave', 'nombre' y 'poblacion' (por ese orden). Ahora hemos insertado tres filas, en las que hemos omitido la clave, y hemos alterado el orden de 'nombre' y 'poblacion'. El valor de la 'clave' se calcula automáticamente, ya que lo hemos definido

como auto-incrementado.

Existe otra sintaxis alternativa, que consiste en indicar el valor para cada columna:

```
mysql> INSERT INTO ciudad5
-> SET nombre='Roma', poblacion=8000000;
Query OK, 1 row affected (0.05 sec)

mysql> SELECT * FROM ciudad5;
+-----+-----+-----+
| clave | nombre | poblacion |
+-----+-----+-----+
|      1 | Madrid | 7000000  |
|      2 | París  | 9000000  |
|      3 | Berlín | 3500000  |
|      4 | Roma   | 8000000  |
+-----+-----+-----+
4 rows in set (0.03 sec)

mysql>
```

Una vez más, a las columnas para las que no indiquemos valores se les asignarán sus valores por defecto. También podemos hacer esto usando el valor *DEFAULT*.

Para las sintaxis que lo permiten, podemos observar que cuando se inserta más de una fila en una única sentencia, obtenemos un mensaje desde **MySQL** que indica el número de filas afectadas, el número de filas duplicadas y el número de avisos.

Para que una fila se considere duplicada debe tener el mismo valor que una fila existente para una clave principal o para una clave única. En tablas en las que no exista clave primaria ni índices de clave única no tiene sentido hablar de filas duplicadas. Es más, en esas tablas es perfectamente posible que existan filas con los mismos valores para todas las columnas.

Por ejemplo, en *mitabla5* tenemos una clave única sobre la columna 'nombre':

```
mysql> INSERT INTO mitabla5 (id, nombre) VALUES
-> (1, 'Carlos'),
-> (2, 'Felipe'),
-> (3, 'Antonio'),
-> (4, 'Carlos'),
-> (5, 'Juan');
ERROR 1062 (23000): Duplicate entry 'Carlos' for key 1
```

```
mysql>
```

Si intentamos insertar dos filas con el mismo valor de la clave única se produce un error y la sentencia no se ejecuta. Pero existe una opción que podemos usar para los casos de claves duplicadas: *ON DUPLICATE KEY UPDATE*. En este caso podemos indicar a **MySQL** qué debe hacer si se intenta insertar una fila que ya existe en la tabla. Las opciones son limitadas: no podemos insertar la nueva fila, sino únicamente modificar la que ya existe. Por ejemplo, en la tabla 'ciudad3' podemos usar el último valor de población en caso de repetición:

```
mysql> INSERT INTO ciudad3 (nombre, poblacion) VALUES
-> ('Madrid', 7000000);
Query OK, 1 rows affected (0.02 sec)

mysql> INSERT INTO ciudad3 (nombre, poblacion) VALUES
-> ('París', 9000000),
-> ('Madrid', 7200000)
-> ON DUPLICATE KEY UPDATE poblacion=VALUES(poblacion);
Query OK, 3 rows affected (0.06 sec)
Records: 2 Duplicates: 1 Warnings: 0

mysql> SELECT * FROM ciudad3;
+-----+-----+
| nombre | poblacion |
+-----+-----+
| Madrid | 7200000 |
| París  | 9000000 |
+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

En este ejemplo, la segunda vez que intentamos insertar la fila correspondiente a 'Madrid' se usará el nuevo valor de población. Si en lugar de *VALUES(poblacion)* usamos *poblacion* el nuevo valor de población se ignora. También podemos usar cualquier expresión:

```
mysql> INSERT INTO ciudad3 (nombre, poblacion) VALUES
-> ('París', 9100000)
-> ON DUPLICATE KEY UPDATE poblacion=poblacion;
Query OK, 2 rows affected (0.02 sec)

mysql> SELECT * FROM ciudad3;
+-----+-----+
| nombre | poblacion |
+-----+-----+
```

```

+-----+-----+
| Madrid | 7200000 |
| París  | 9000000 |
+-----+-----+
2 rows in set (0.00 sec)

mysql> INSERT INTO ciudad3 (nombre, poblacion) VALUES
-> ('París', 9100000)
-> ON DUPLICATE KEY UPDATE poblacion=0;
Query OK, 2 rows affected (0.01 sec)

mysql> SELECT * FROM ciudad3;
+-----+-----+
| nombre | poblacion |
+-----+-----+
| Madrid | 7200000 |
| París  | 0 |
+-----+-----+
2 rows in set (0.00 sec)

mysql>

```

## Reemplazar filas

Existe una sentencia **REPLACE**, que es una alternativa para **INSERT**, que sólo se diferencia en que si existe algún registro anterior con el mismo valor para una clave primaria o única, se elimina el viejo y se inserta el nuevo en su lugar.

```

REPLACE [LOW_PRIORITY | DELAYED]
  [INTO] tbl_name [(col_name,...)]
  VALUES ({expr | DEFAULT},...),(...),...

```

```

REPLACE [LOW_PRIORITY | DELAYED]
  [INTO] tbl_name
  SET col_name={expr | DEFAULT}, ...

```

```

mysql> REPLACE INTO ciudad3 (nombre, poblacion) VALUES
-> ('Madrid', 7200000),
-> ('París', 9200000),
-> ('Berlín', 6000000);
Query OK, 5 rows affected (0.05 sec)

```

```
Records: 3 Duplicates: 2 Warnings: 0
```

```
mysql> SELECT * FROM ciudad3;
```

```
+-----+-----+
| nombre | poblacion |
+-----+-----+
| Berlín | 6000000 |
| Madrid | 7200000 |
| París  | 9200000 |
+-----+-----+
3 rows in set (0.00 sec)
```

```
mysql>
```

En este ejemplo se sustituyen las filas correspondientes a 'Madrid' y 'París', que ya existían en la tabla y se inserta la de 'Berlín' que no estaba previamente.

Las mismas sintaxis que existen para [INSERT](#), están disponibles para [REPLACE](#):

```
mysql> REPLACE INTO ciudad3 VALUES ('Roma', 9500000);
Query OK, 1 rows affected (0.03 sec)
```

```
mysql> REPLACE INTO ciudad3 SET nombre='Londres', poblacion=10000000;
Query OK, 1 row affected (0.03 sec)
```

```
mysql> SELECT * FROM ciudad3;
```

```
+-----+-----+
| nombre | poblacion |
+-----+-----+
| Berlín | 6000000 |
| Londres | 10000000 |
| Madrid | 7200000 |
| París  | 9200000 |
| Roma   | 9500000 |
+-----+-----+
5 rows in set (0.00 sec)
```

```
mysql>
```

## Actualizar filas

Podemos modificar valores de las filas de una tabla usando la sentencia [UPDATE](#). En su forma más simple, los cambios se aplican a todas las filas, y a las columnas que especifiquemos.

```
UPDATE [LOW_PRIORITY] [IGNORE] tbl_name
  SET col_name1=expr1 [, col_name2=expr2 ...]
  [WHERE where_definition]
  [ORDER BY ...]
  [LIMIT row_count]
```

Por ejemplo, podemos aumentar en un 10% la población de todas las ciudades de la tabla ciudad3 usando esta sentencia:

```
mysql> UPDATE ciudad3 SET poblacion=poblacion*1.10;
Query OK, 5 rows affected (0.15 sec)
Rows matched: 5  Changed: 5  Warnings: 0
```

```
mysql> SELECT * FROM ciudad3;
```

```
+-----+-----+
| nombre | poblacion |
+-----+-----+
| Berlín | 6600000 |
| Londres | 11000000 |
| Madrid | 7920000 |
| París | 10120000 |
| Roma | 10450000 |
+-----+-----+
```

```
5 rows in set (0.00 sec)
```

```
mysql>
```

Podemos, del mismo modo, actualizar el valor de más de una columna, separandolas en la sección *SET* mediante comas:

```
mysql> UPDATE ciudad5 SET clave=clave+10, poblacion=poblacion*0.97;
Query OK, 4 rows affected (0.05 sec)
Rows matched: 4  Changed: 4  Warnings: 0
```

```
mysql> SELECT * FROM ciudad5;
```

```
+-----+-----+-----+
| clave | nombre | poblacion |
+-----+-----+-----+
| 11 | Madrid | 6790000 |
| 12 | París | 8730000 |
| 13 | Berlín | 3395000 |
+-----+-----+-----+
```

```
|      14 | Roma      |      7760000 |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

En este ejemplo hemos incrementado el valor de la columna 'clave' en 10 y disminuido el de la columna 'poblacion' en un 3%, para todas las filas.

Pero no tenemos por qué actualizar todas las filas de la tabla. Podemos limitar el número de filas afectadas de varias formas.

La primera es mediante la cláusula *WHERE*. Usando esta cláusula podemos establecer una condición. Sólo las filas que cumplan esa condición serán actualizadas:

```
mysql> UPDATE ciudad5 SET poblacion=poblacion*1.03
      -> WHERE nombre='Roma';
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM ciudad5;
+-----+-----+-----+
| clave | nombre | poblacion |
+-----+-----+-----+
|      11 | Madrid | 6790000 |
|      12 | París  | 8730000 |
|      13 | Berlín | 3395000 |
|      14 | Roma   | 7992800 |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

En este caso sólo hemos aumentado la población de las ciudades cuyo nombre sea 'Roma'. Las condiciones pueden ser más complejas. Existen muchas funciones y operadores que se pueden aplicar sobre cualquier tipo de columna, y también podemos usar operadores booleanos como *AND* u *OR*. Veremos esto con más detalle en otros capítulos.

Otra forma de limitar el número de filas afectadas es usar la cláusula *LIMIT*. Esta cláusula permite especificar el número de filas a modificar:

```
mysql> UPDATE ciudad5 SET clave=clave-10 LIMIT 2;
```

```
Query OK, 2 rows affected (0.05 sec)
Rows matched: 2  Changed: 2  Warnings: 0
```

```
mysql> SELECT * FROM ciudad5;
+-----+-----+-----+
| clave | nombre | poblacion |
+-----+-----+-----+
|      1 | Madrid | 6790000  |
|      2 | París  | 8730000  |
|     13 | Berlín | 3395000  |
|     14 | Roma   | 7992800  |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

En este ejemplo hemos decrementado en 10 unidades la columna clave de las dos primeras filas.

Esta cláusula se puede combinar con *WHERE*, de modo que sólo las 'n' primeras filas que cumplan una determinada condición se modifiquen.

Sin embargo esto no es lo habitual, ya que, si no existen claves primarias o únicas, el orden de las filas es arbitrario, no tiene sentido seleccionarlas usando sólo la cláusula *LIMIT*.

La cláusula *LIMIT* se suele asociar a la cláusula *ORDER BY*. Por ejemplo, si queremos modificar la fila con la fecha más antigua de la tabla 'gente', usaremos esta sentencia:

```
mysql> UPDATE gente SET fecha="1985-04-12" ORDER BY fecha LIMIT 1;
Query OK, 1 row affected, 1 warning (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 1
```

```
mysql> SELECT * FROM gente;
+-----+-----+
| nombre | fecha      |
+-----+-----+
| Fulano  | 1985-04-12 |
| Mengano | 1978-06-15 |
| Tulano  | 2000-12-02 |
| Pegano  | 1993-02-10 |
+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

Si queremos modificar la fila con la fecha más reciente, usaremos el orden inverso, es decir, el descendente:

```
mysql> UPDATE gente SET fecha="2001-12-02" ORDER BY fecha DESC LIMIT 1;
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM gente;
+-----+-----+
| nombre | fecha      |
+-----+-----+
| Fulano  | 1985-04-12 |
| Mengano | 1978-06-15 |
| Tulano  | 2001-12-02 |
| Pegano  | 1993-02-10 |
+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

Cuando exista una clave primaria o única, se usará ese orden por defecto, si no se especifica una cláusula *ORDER BY*.

## Eliminar filas

Para eliminar filas se usa la sentencia [DELETE](#). La sintaxis es muy parecida a la de [UPDATE](#):

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM table_name
      [WHERE where_definition]
      [ORDER BY ...]
      [LIMIT row_count]
```

La forma más simple es no usar ninguna de las cláusulas opcionales:

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

mysql>
```

De este modo se eliminan todas las filas de la tabla.

Pero es más frecuente que sólo queramos eliminar ciertas filas que cumplan determinadas condiciones. La forma más normal de hacer esto es usar la cláusula *WHERE*:

```
mysql> DELETE FROM ciudad5 WHERE clave=2;
Query OK, 1 row affected (0.05 sec)
```

```
mysql> SELECT * FROM ciudad5;
+-----+-----+-----+
| clave | nombre | poblacion |
+-----+-----+-----+
|      1 | Madrid | 6790000 |
|     13 | Berlín | 3395000 |
|     14 | Roma   | 7992800 |
+-----+-----+-----+
3 rows in set (0.01 sec)
```

```
mysql>
```

También podemos usar las cláusulas *LIMIT* y *ORDER BY* del mismo modo que en la sentencia [UPDATE](#), por ejemplo, para eliminar las dos ciudades con más población:

```
mysql> DELETE FROM ciudad5 ORDER BY poblacion DESC LIMIT 2;
Query OK, 2 rows affected (0.03 sec)
```

```
mysql> SELECT * FROM ciudad5;
+-----+-----+-----+
| clave | nombre | poblacion |
+-----+-----+-----+
|     13 | Berlín | 3395000 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql>
```

## Vaciar una tabla

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

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

```
mysql> TRUNCATE ciudad5;  
Query OK, 1 row affected (0.05 sec)
```

```
mysql>
```

## 9 Lenguaje SQL

### Selección de datos

Ya disponemos de bases de datos, y sabemos cómo añadir y modificar datos. Ahora aprenderemos a extraer datos de una base de datos. Para ello volveremos a usar la sentencia SELECT.

La sintaxis de SELECT es compleja, pero en este capítulo no explicaremos todas sus opciones. Una forma más general consiste en la siguiente sintaxis:

```
SELECT [ALL | DISTINCT | DISTINCTROW]
  expresion_select,...
FROM referencias_de_tablas
WHERE condiciones
[GROUP BY {nombre_col | expresion | posicion}
  [ASC | DESC], ... [WITH ROLLUP]]
[HAVING condiciones]
[ORDER BY {nombre_col | expresion | posicion}
  [ASC | DESC] ,...]
[LIMIT {[desplazamiento,] contador | contador OFFSET desplazamiento}]
```

#### Forma incondicional

La forma más sencilla es la que hemos usado hasta ahora, consiste en pedir todas las columnas y no especificar condiciones.

```
mysql>mysql> SELECT * FROM gente;
```

```
+-----+-----+
| nombre | fecha      |
+-----+-----+
| Fulano  | 1985-04-12 |
| Mengano | 1978-06-15 |
| Tulano  | 2001-12-02 |
| Pegano  | 1993-02-10 |
+-----+-----+
4 rows in set (0.00 sec)
```

```
mysql>
```

## Limitar las columnas: proyección

Recordemos que una de las operaciones del álgebra relacional era la proyección, que consistía en seleccionar determinados atributos de una relación.

Mediante la sentencia SELECT es posible hacer una proyección de una tabla, seleccionando las columnas de las que queremos obtener datos. En la sintaxis que hemos mostrado, la selección de columnas corresponde con la parte "expresion\_select". En el ejemplo anterior hemos usado '\*', que quiere decir que se muestran todas las columnas.

Pero podemos usar una lista de columnas, y de ese modo sólo se mostrarán esas columnas:

```
mysql> SELECT nombre FROM gente;
+-----+
| nombre |
+-----+
| Fulano |
| Mengano |
| Tulano |
| Pegano |
+-----+
4 rows in set (0.00 sec)

mysql> SELECT clave,poblacion FROM ciudad5;
Empty set (0.00 sec)

mysql>
```

Las expresiones\_select no se limitan a nombres de columnas de tablas, pueden ser otras expresiones, incluso aunque no correspondan a ninguna tabla:

```
mysql> SELECT SIN(3.1416/2), 3+5, 7*4;
+-----+-----+-----+
| SIN(3.1416/2) | 3+5 | 7*4 |
+-----+-----+-----+
| 0.99999999999325 | 8 | 28 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Vemos que podemos usar funciones, en este ejemplo hemos usando la función [SIN](#) para calcular el seno de  $\pi/2$ . En próximos capítulos veremos muchas de las funciones de las que disponemos en **MySQL**.

También podemos aplicar funciones sobre columnas de tablas, y usar esas columnas en expresiones para generar nuevas columnas:

```
mysql> SELECT nombre, fecha, DATEDIFF(CURRENT_DATE(), fecha)/365 FROM
gente;
```

nombre	fecha	DATEDIFF(CURRENT_DATE(), fecha)/365
Fulano	1985-04-12	19.91
Mengano	1978-06-15	26.74
Tulano	2001-12-02	3.26
Pegano	1993-02-10	12.07

```
4 rows in set (0.00 sec)
```

```
mysql>
```

## Alias

Aprovechemos la ocasión para mencionar que también es posible asignar un alias a cualquiera de las expresiones select. Esto se puede hacer usando la palabra **AS**, aunque esta palabra es opcional:

```
mysql> SELECT nombre, fecha, DATEDIFF(CURRENT_DATE(), fecha)/365 AS edad
-> FROM gente;
```

nombre	fecha	edad
Fulano	1985-04-12	19.91
Mengano	1978-06-15	26.74
Tulano	2001-12-02	3.26
Pegano	1993-02-10	12.07

```
4 rows in set (0.00 sec)
```

```
mysql>
```

Podemos hacer "bromas" como:

```
mysql> SELECT 2+3 "2+2";
+-----+
| 2+2 |
+-----+
|    5 |
+-----+
1 row in set (0.00 sec)

mysql>
```

En este caso vemos que podemos omitir la palabra *AS*. Pero no es aconsejable, ya que en ocasiones puede ser difícil distinguir entre un olvido de una coma o de una palabra *AS*.

Posteriormente veremos que podemos usar los alias en otras cláusulas, como *WHERE*, *HAVING* o *GROUP BY*.

## Mostrar filas repetidas

Ya que podemos elegir sólo algunas de las columnas de una tabla, es posible que se produzcan filas repetidas, debido a que hayamos excluido las columnas únicas.

Por ejemplo, añadamos las siguientes filas a nuestra tabla:

```
mysql> INSERT INTO gente VALUES ('Pimplano', '1978-06-15'),
-> ('Frutano', '1985-04-12');
Query OK, 2 rows affected (0.03 sec)
Records: 2 Duplicates: 0 Warnings: 0

mysql> SELECT fecha FROM gente;
+-----+
| fecha          |
+-----+
| 1985-04-12    |
| 1978-06-15    |
| 2001-12-02    |
| 1993-02-10    |
| 1978-06-15    |
| 1985-04-12    |
+-----+
6 rows in set (0.00 sec)

mysql>
```

Vemos que existen dos valores de filas repetidos, para la fecha "1985-04-12" y para "1978-06-15". La sentencia que hemos usado asume el valor por defecto (*ALL*) para el grupo de opciones *ALL*, *DISTINCT* y *DISTINCTROW*. En realidad sólo existen dos opciones, ya que las dos últimas: *DISTINCT* y *DISTINCTROW* son sinónimos.

La otra alternativa es usar *DISTINCT*, que hará que sólo se muestren las filas diferentes:

```
mysql> SELECT DISTINCT fecha FROM gente;
+-----+
| fecha      |
+-----+
| 1985-04-12 |
| 1978-06-15 |
| 2001-12-02 |
| 1993-02-10 |
+-----+
4 rows in set (0.00 sec)

mysql>
```

## Limitar las filas: selección

Otra de las operaciones del álgebra relacional era la selección, que consistía en seleccionar filas de una realación que cumplieran determinadas condiciones.

Lo que es más útil de una base de datos es la posibilidad de hacer consultas en función de ciertas condiciones. Generalmente nos interesará saber qué filas se ajustan a determinados parámetros. Por supuesto, SELECT permite usar condiciones como parte de su sintaxis, es decir, para hacer selecciones.

Concretamente mediante la cláusula *WHERE*, veamos algunos ejemplos:

```
mysql> SELECT * FROM gente WHERE nombre="Mengano";
+-----+-----+
| nombre  | fecha      |
+-----+-----+
| Mengano | 1978-06-15 |
+-----+-----+
1 row in set (0.03 sec)

mysql> SELECT * FROM gente WHERE fecha>="1986-01-01";
+-----+-----+
| nombre  | fecha      |
+-----+-----+
```

```

+-----+-----+
| Tulano | 2001-12-02 |
| Pegano | 1993-02-10 |
+-----+-----+
2 rows in set (0.00 sec)

mysql> SELECT * FROM gente
      -> WHERE fecha>="1986-01-01" AND fecha < "2000-01-01";
+-----+-----+
| nombre | fecha      |
+-----+-----+
| Pegano | 1993-02-10 |
+-----+-----+
1 row in set (0.00 sec)

mysql>

```

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

También se puede aplicar lógica booleana para crear expresiones complejas. Disponemos de los operadores *AND*, *OR*, *XOR* y *NOT*.

En próximos capítulos veremos los operadores de los que dispone **MySQL**.

## Agrupar filas

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

```

mysql> SELECT fecha FROM gente GROUP BY fecha;
+-----+
| fecha      |
+-----+
| 1978-06-15 |
| 1985-04-12 |
| 1993-02-10 |
| 2001-12-02 |
+-----+
4 rows in set (0.00 sec)

```

```
mysql>
```

La primera diferencia que observamos es que si se usa *GROUP BY* la salida se ordena según los valores de la columna indicada. En este caso, las columnas aparecen ordenadas por fechas.

Otra diferencia es que se eliminan los valores duplicados aún si la proyección no contiene filas duplicadas, por ejemplo:

```
mysql> SELECT nombre, fecha FROM gente GROUP BY fecha;
```

```
+-----+-----+
| nombre | fecha      |
+-----+-----+
| Mengano | 1978-06-15 |
| Fulano  | 1985-04-12 |
| Pegano  | 1993-02-10 |
| Tulano  | 2001-12-02 |
+-----+-----+
4 rows in set (0.00 sec)
```

```
mysql>
```

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

```
mysql> SELECT fecha, COUNT(*) AS cuenta FROM gente GROUP BY fecha;
```

```
+-----+-----+
| fecha      | cuenta |
+-----+-----+
| 1978-06-15 |      2 |
| 1985-04-12 |      2 |
| 1993-02-10 |      1 |
| 2001-12-02 |      1 |
+-----+-----+
4 rows in set (0.00 sec)
```

```
mysql>
```

Esta sentencia muestra todas las fechas diferentes y el número de filas para cada fecha.

Existen otras funciones de resumen o reunión, como [MAX\(\)](#), [MIN\(\)](#), [SUM\(\)](#), [AVG\(\)](#), [STD\(\)](#), [VARIANCE\(\)](#)...

Estas funciones también se pueden usar sin la cláusula *GROUP BY* siempre que no se proyecten otras columnas:

```
mysql> SELECT MAX(nombre) FROM gente;
+-----+
| max(nombre) |
+-----+
| Tulano      |
+-----+
1 row in set (0.00 sec)

mysql>
```

Esta sentencia muestra el valor más grande de 'nombre' de la tabla 'gente', es decir, el último por orden alfabético.

## Cláusula HAVING

La cláusula *HAVING* permite hacer selecciones en situaciones en las que no es posible usar *WHERE*. Veamos un ejemplo completo:

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

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

mysql> SELECT ciudad, MAX(temperatura) FROM muestras
-> GROUP BY ciudad HAVING MAX(temperatura)>16;
+-----+-----+
| ciudad | MAX(temperatura) |
+-----+-----+
```

```

| Berlín | 18 |
| Madrid | 25 |
+-----+-----+
2 rows in set (0.00 sec)

mysql>

```

La cláusula *WHERE* no se puede aplicar a columnas calculadas mediante funciones de reunión, como en este ejemplo.

## Ordenar resultados

Además, podemos añadir una cláusula de orden *ORDER BY* para obtener resultados ordenados por la columna que queramos:

```

mysql> SELECT * FROM gente ORDER BY fecha;
+-----+-----+
| nombre | fecha      |
+-----+-----+
| Mengano | 1978-06-15 |
| Pimplano | 1978-06-15 |
| Fulano   | 1985-04-12 |
| Frutano  | 1985-04-12 |
| Pegano   | 1993-02-10 |
| Tulano   | 2001-12-02 |
+-----+-----+
6 rows in set (0.02 sec)

mysql>

```

Existe una opción para esta cláusula para elegir el orden, ascendente o descendente. Se puede añadir a continuación *ASC* o *DESC*, respectivamente. Por defecto se usa el orden ascendente, de modo que el modificador *ASC* es opcional.

```

mysql> SELECT * FROM gente ORDER BY fecha DESC;
+-----+-----+
| nombre | fecha      |
+-----+-----+
| Tulano  | 2001-12-02 |
| Pegano  | 1993-02-10 |
| Fulano  | 1985-04-12 |
| Frutano | 1985-04-12 |

```

```

| Mengano | 1978-06-15 |
| Pimplano | 1978-06-15 |
+-----+-----+
6 rows in set (0.00 sec)

mysql>

```

## Limitar el número de filas de salida

Por último, la cláusula *LIMIT* permite limitar el número de filas devueltas:

```

mysql> SELECT * FROM gente LIMIT 3;
+-----+-----+
| nombre | fecha      |
+-----+-----+
| Fulano | 1985-04-12 |
| Mengano | 1978-06-15 |
| Tulano | 2001-12-02 |
+-----+-----+
3 rows in set (0.19 sec)

mysql>

```

Esta cláusula se suele usar para obtener filas por grupos, y no sobrecargar demasiado al servidor, o a la aplicación que recibe los resultados. Para poder hacer esto la cláusula *LIMIT* admite dos parámetros. Cuando se usan los dos, el primero indica el número de la primera fila a recuperar, y el segundo el número de filas a recuperar. Podemos, por ejemplo, recuperar las filas de dos en dos:

```

mysql> Select * from gente limit 0,2;
+-----+-----+
| nombre | fecha      |
+-----+-----+
| Fulano | 1985-04-12 |
| Mengano | 1978-06-15 |
+-----+-----+
2 rows in set (0.00 sec)

mysql> Select * from gente limit 2,2;
+-----+-----+
| nombre | fecha      |
+-----+-----+
| Tulano | 2001-12-02 |

```

```
| Pegano | 1993-02-10 |  
+-----+-----+  
2 rows in set (0.02 sec)
```

```
mysql> Select * from gente limit 4,2;
```

```
+-----+-----+  
| nombre | fecha |  
+-----+-----+  
| Pimplano | 1978-06-15 |  
| Frutano | 1985-04-12 |  
+-----+-----+  
2 rows in set (0.00 sec)
```

```
mysql> Select * from gente limit 6,2;  
Empty set (0.00 sec)
```

```
mysql>
```

# 10 Lenguaje SQL

## Operadores

**MySQL** dispone de multitud de operadores diferentes para cada uno de los tipos de columna. Esos operadores se utilizan para construir expresiones que se usan en cláusulas *ORDER BY* y *HAVING* de la sentencia [SELECT](#) y en las cláusulas *WHERE* de las sentencias [SELECT](#), [DELETE](#) y [UPDATE](#). Además se pueden emplear en sentencias [SET](#).

### Operador de asignación

En **MySQL** podemos crear variables y usarlas posteriormente en expresiones.

Para crear una variable hay dos posibilidades. La primera consiste en usar la sentencia [SET](#) de este modo:

```
mysql> SET @hoy = CURRENT_DATE();
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> SELECT @hoy;
+-----+
| @hoy   |
+-----+
| 2005-03-23 |
+-----+
1 row in set (0.00 sec)
```

```
mysql>
```

La otra alternativa permite definir variables de usuario dentro de una sentencia [SELECT](#):

```
mysql> SELECT @x:=10;
+-----+
| @x:=10 |
+-----+
|      10 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT @x;
```

```
+-----+
| @x    |
+-----+
| 10    |
+-----+
1 row in set (0.00 sec)

mysql>
```

En esta segunda forma es donde se usa el operador de asignación `:=`. Otros ejemplos del uso de variables de usuario pueden ser:

```
mysql> SELECT @fecha_min:=MIN(fecha), @fecha_max:=MAX(fecha) FROM gente;
+-----+-----+
| @fecha_min:=MIN(fecha) | @fecha_max:=MAX(fecha) |
+-----+-----+
| 1978-06-15            | 2001-12-02            |
+-----+-----+
1 row in set (0.00 sec)

mysql> SELECT * FROM gente WHERE fecha=@fecha_min;
+-----+-----+
| nombre  | fecha      |
+-----+-----+
| Mengano | 1978-06-15 |
| Pimplano | 1978-06-15 |
+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

Una variable sin asignar será de tipo cadena y tendrá el valor *NULL*.

## Operadores lógicos

Los operadores lógicos se usan para crear expresiones lógicas complejas. Permiten el uso de álgebra booleana, y nos ayudarán a crear condiciones mucho más precisas.

En el álgebra booleana sólo existen dos valores posibles para los operandos y los resultados: verdadero y falso. **MySQL** dispone de dos constantes para esos valores: *TRUE* y *FALSE*, respectivamente.

**MySQL** añade un tercer valor: desconocido. Esto es para que sea posible trabajar con valores *NULL*.

El valor verdadero se implementa como 1 o *TRUE*, el falso como 0 o *FALSE* y el desconocido como *NULL*.

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

mysql>
```

## Operador Y

En **MySQL** se puede usar tanto la forma *AND* como *&&*, es decir, ambas formas se refieren al mismo operador: Y lógico.

Se trata de un operador binario, es decir, requiere de dos operandos. El resultado es verdadero sólo si ambos operandos son verdaderos, y falso si cualquier operando es falso. Esto se representa mediante la siguiente tabla de verdad:

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

Al igual que todos los operadores binarios que veremos, el operador Y se puede asociar, es decir, se pueden crear expresiones como *A AND B AND C*. El hecho de que se requieran dos operandos significa que las operaciones se realizan tomando los operandos dos a dos, y estas expresiones se evalúan de izquierda a derecha. Primero se evalúa *A AND B*, y el resultado, *R*, se usa como primer operando de la siguiente operación *R AND C*.

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

mysql>
```

## Operador O

En **MySQL** este operador también tiene dos formas equivalentes *OR* y *//*

El operador **O** también es binario. Si ambos operandos son distintos de *NULL* y el resultado es verdadero si cualquiera de ellos es verdadero, y falso si ambos son falsos. Si uno de los operandos es *NULL* el resultado es verdadero si el otro es verdadero, y *NULL* en el caso contrario. La tabla de verdad es:

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

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

mysql>
```

## Operador O exclusivo

*XOR* también es un operador binario, que devuelve *NULL* si cualquiera de los operandos es *NULL*. Si ninguno de los operandos es *NULL* devolverá un valor verdadero si uno de ellos es verdadero, y falso si ambos son verdaderos o ambos falsos. La tabla de verdad será:

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

```
mysql> SELECT 1 XOR 0, 1 XOR NULL, 0 XOR NULL, 1 XOR 0 XOR 1;
```

```
+-----+-----+-----+-----+
| 1 XOR 0 | 1 XOR NULL | 0 XOR NULL | 1 XOR 0 XOR 1 |
+-----+-----+-----+-----+
|          1 |          NULL |          NULL |                  0 |
+-----+-----+-----+-----+
```

```
1 row in set (0.00 sec)
```

```
mysql>
```

## Operador de negación

El operador *NOT*, que también se puede escribir como *!*, es un operador unitario, es decir sólo afecta a un operando. Si el operando es verdadero devuelve falso, y viceversa. Si el operando es *NULL* el valor devuelto también es *NULL*.

A	NOT A
falso	verdadero
verdadero	falso

NULL	NULL
------	------

```
mysql> SELECT NOT 0, NOT 1, NOT NULL;
+-----+-----+-----+
| NOT 0 | NOT 1 | NOT NULL |
+-----+-----+-----+
|      1 |      0 |      NULL |
+-----+-----+-----+
1 row in set (0.02 sec)

mysql>
```

## Reglas para las comparaciones de valores

**MySQL** Sigue las siguientes reglas a la hora de comparar valores:

- Si uno o los dos valores a comparar son *NULL*, el resultado es *NULL*, excepto con el operador  $\lt;=\gt;$ , de comparación con *NULL* segura.
- Si los dos valores de la comparación son cadenas, se comparan como cadenas.
- Si ambos valores son enteros, se comparan como enteros.
- Los valores hexadecimales se tratan como cadenas binarias, si no se comparan con un número.
- Si uno de los valores es del tipo *TIMESTAMP* o *DATETIME* y el otro es una constante, la constantes se convierte a *timestamp* antes de que se lleve a cabo la comparación. Hay que tener en cuenta que esto no se hace para los argumentos de una expresión *IN()*. Para estar seguro, es mejor usar siempre cadenas completas *datetime/date/time strings* cuando se hacen comparaciones.
- En el resto de los casos, los valores se comparan como números en coma flotante.

## Operadores de comparación

Para crear expresiones lógicas, a las que podremos aplicar el álgebra de Boole, disponemos de varios operadores de comparación. Estos operadores se aplican a cualquier tipo de columna: fechas, cadenas, números, etc, y devuelven valores lógicos: verdadero o falso (1/0).

Los operadores de comparación son los habituales en cualquier lenguaje de programación, pero además, **MySQL** añade varios más que resultan de mucha utilidad, ya que son de uso muy frecuente.

### Operador de igualdad

El operador  $\lt;=\gt;$  compara dos expresiones, y da como resultado 1 si son iguales, o 0 si son diferentes. Ya lo hemos usado en ejemplos anteriormente:

```
mysql> SELECT * FROM gente WHERE fecha="2001-12-02";
+-----+-----+
| nombre | fecha      |
+-----+-----+
| Tulano | 2001-12-02 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Hay que mencionar que, al contrario que otros lenguajes, como C o C++, donde el control de tipos es muy estricto, en **MySQL** se pueden comparar valores de tipos diferentes, y el resultado será el esperado.

Por ejemplo:

```
mysql> SELECT "0" = 0, "0.1"=.1;
+-----+-----+
| "0" = 0 | "0.1"=.1 |
+-----+-----+
|          |          |
|          1 |          1 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Esto es así porque **MySQL** hace conversión de tipos de forma implícita, incluso cuando se trate de valores de tipo cadena.

## Operador de igualdad con *NULL* seguro

El operador `<=>` funciona igual que el operador `=`, salvo que si en la comparación una o ambas de las expresiones es nula el resultado no es *NULL*. Si se comparan dos expresiones nulas, el resultado es verdadero:

```
mysql> SELECT NULL = 1, NULL = NULL;
+-----+-----+
| NULL = 1 | NULL = NULL |
+-----+-----+
|          |          |
|          NULL |          NULL |
+-----+-----+
```

```

1 row in set (0.00 sec)

mysql> SELECT NULL <=> 1, NULL <=> NULL;
+-----+-----+
| NULL <=> 1 | NULL <=> NULL |
+-----+-----+
|          0 |             1 |
+-----+-----+
1 row in set (0.00 sec)

mysql>

```

## Operador de desigualdad

**MySQL** dispone de dos operadores equivalente para comprobar desigualdades,  $<>$  y  $\neq$ . Si las expresiones comparadas son diferentes, el resultado es verdadero, y si son iguales, el resultado es falso:

```

mysql> SELECT 100 <> 32, 43 != 43;
+-----+-----+
| 100 <> 32 | 43 != 43 |
+-----+-----+
|          1 |          0 |
+-----+-----+
1 row in set (0.02 sec)

mysql>

```

## Operadores de comparación de magnitud

Disponemos de los cuatro operadores corrientes.

Operador	Descripción
$<=$	Menor o igual
$<$	Menor
$>$	Mayor
$>=$	Mayor o igual

Estos operadores también permiten comparar cadenas, fechas, y por supuesto, números:

```
mysql> SELECT "hola" < "adios", "2004-12-31" > "2004-12-01";
+-----+-----+
| "hola" < "adios" | "2004-12-31" > "2004-12-01" |
+-----+-----+
|                0 |                1 |
+-----+-----+
1 row in set (0.00 sec)

mysql> SELECT ".01" >= "0.01", .01 >= 0.01;
+-----+-----+
| ".01" >= "0.01" | .01 >= 0.01 |
+-----+-----+
|                0 |                1 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Cuando se comparan cadenas, se considerará menor la cadena que aparezca antes por orden alfabético.

Si son fechas, se considera que es menor cuanto más antigua sea.

Pero cuidado, como vemos en el segundo ejemplo, si comparamos cadenas que contienen números, no hay conversión, y se comparan las cadenas tal como aparecen.

## Verificación de *NULL*

Los operadores *IS NULL* e *IS NOT NULL* sirven para verificar si una expresión determinada es o no nula. La sintaxis es:

```
<expresión> IS NULL
<expresión> IS NOT NULL
```

Por ejemplo:

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

```
mysql> SELECT "NULL" IS NOT NULL;
+-----+
| "NULL" IS NOT NULL |
+-----+
|                1 |
+-----+
1 row in set (0.00 sec)

mysql>
```

## Verificar pertenencia a un rango

Entre los operadores de **MySQL**, hay uno para comprobar si una expresión está comprendida en un determinado rango de valores. La sintaxis es:

```
<expresión> BETWEEN mínimo AND máximo
<expresión> NOT BETWEEN mínimo AND máximo
```

En realidad es un operador prescindible, ya que se puede usar en su lugar dos expresiones de comparación y el operador **AND**. Estos dos ejemplos son equivalentes:

```
mysql> SELECT 23 BETWEEN 1 AND 100;
+-----+
| 23 BETWEEN 1 AND 100 |
+-----+
|                1 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT 23 >= 1 AND 23 <= 100;
+-----+
| 23 >= 1 AND 23 <= 100 |
+-----+
|                1 |
+-----+
1 row in set (0.00 sec)

mysql>
```

Del mismo modo, estas dos expresiones también lo son:

```
mysql> SELECT 23 NOT BETWEEN 1 AND 100;
+-----+
| 23 NOT BETWEEN 1 AND 100 |
+-----+
|                               0 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT NOT (23 BETWEEN 1 AND 100);
+-----+
| NOT (23 BETWEEN 1 AND 100) |
+-----+
|                               0 |
+-----+
1 row in set (0.00 sec)

mysql>
```

## Elección de no nulos

El operador *COALESCE* sirve para seleccionar el primer valor no nulo de una lista o conjunto de expresiones. La sintaxis es:

```
COALESCE(<expr1>, <expr2>, <expr3>...)
```

El resultado es el valor de la primera expresión distinta de *NULL* que aparezca en la lista. Por ejemplo:

```
mysql> SET @a=23, @b="abc", @d="1998-11-12";
Query OK, 0 rows affected (0.03 sec)

mysql> SELECT COALESCE(@c, @a, @b, @d);
+-----+
| COALESCE(@c, @a, @b, @d) |
+-----+
| 23                        |
+-----+
1 row in set (0.05 sec)

mysql>
```

En este ejemplo no hemos definido la variable @c, por lo tanto, tal como dijimos antes, su valor se considera *NULL*. El operador *COALESCE* devuelve el valor de la primera variable no nula, es decir, el valor de @a.

## Valores máximo y mínimo de una lista

Los operadores *GREATEST* y *LEAST* devuelven el valor máximo y mínimo, respectivamente, de la lista de expresiones dada. La sintaxis es:

```
GREATEST(<expr1>, <expr2>, <expr3>...)
LEAST(<expr1>, <expr2>, <expr3>...)
```

La lista de expresiones debe contener al menos dos valores.

Los argumentos se comparan según estas reglas:

- Si el valor de retorno se usa en un contexto entero, o si todos los elementos de la lista son enteros, estos se comparan entre sí como enteros.
- Si el valor de retorno se usa en un contexto real, o si todos los elementos son valores reales, serán comparados como reales.
- Si cualquiera de los argumentos es una cadena sensible al tipo (mayúsculas y minúsculas son caracteres diferentes), los argumentos se comparan como cadenas sensibles al tipo.
- En cualquier otro caso, los argumentos se comparan como cadenas no sensibles al tipo.

```
mysql> SELECT LEAST(2,5,7,1,23,12);
+-----+
| LEAST(2,5,7,1,23,12) |
+-----+
|                1 |
+-----+
1 row in set (0.69 sec)

mysql> SELECT GREATEST(2,5,7,1,23,12);
+-----+
| GREATEST(2,5,7,1,23,12) |
+-----+
|                23 |
+-----+
1 row in set (0.03 sec)

mysql> SELECT GREATEST(2,5,"7",1,"a",12);
+-----+
```



Por ejemplo:

```
mysql> SELECT 1/0 IS NULL, ISNULL(1/0);
+-----+-----+
| 1/0 IS NULL | ISNULL(1/0) |
+-----+-----+
|          1 |          1 |
+-----+-----+
1 row in set (0.02 sec)

mysql>
```

## Encontrar intervalo

Se puede usar el operador *INTERVAL* para calcular el intervalo al que pertenece un valor determinado. La sintaxis es:

```
INTERVAL(<expresión>, <límite1>, <limite1>, ... <limiten>)
```

Si el valor de la expresión es menor que límite1, el operador regresa con el valor 0, si es mayor o igual que límite1 y menor que limite2, regresa con el valor 1, etc.

Todos los valores de los límites deben estar ordenados, ya que **MySQL** usa el algoritmo de búsqueda binaria.

**Nota:** Según la documentación, los valores de los índices se trata siempre como enteros, aunque he podido verificar que el operador funciona también con valores en coma flotante, cadenas y fechas.

```
mysql> SET @x = 19;
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT INTERVAL(@x, 0, 10, 20, 30, 40);
+-----+-----+
| INTERVAL(@x, 0, 10, 20, 30, 40) |
+-----+-----+
|                2 |
+-----+-----+
1 row in set (0.01 sec)
```

```
mysql> SELECT INTERVAL("Gerardo", "Antonio",
-> "Fernando", "Ramón", "Xavier");
+-----+-----+
| INTERVAL("Gerardo", "Antonio", "Fernando", "Ramón", "Xavier") |
+-----+-----+
|                                                                    4 |
+-----+-----+
1 row in set (0.01 sec)

mysql>
```

## Operadores aritméticos

Los operadores aritméticos se aplican a valores numéricos, ya sean enteros o en coma flotante. El resultado siempre es un valor numérico, entero o en coma flotante.

**MySQL** dispone de los operadores aritméticos habituales: suma, resta, multiplicación y división.

En el caso de los operadores de suma, resta, cambio de signo y multiplicación, si los operandos son enteros, el resultado se calcula usando el tipo **BIGINT**, es decir, enteros de 64 bits. Hay que tener esto en cuenta, sobre todo en el caso de números grandes.

### Operador de adición o suma

El operador para la suma es, como cabría esperar, **+**. No hay mucho que comentar al respecto. Por ejemplo:

```
mysql> SELECT 192+342, 23.54+23;
+-----+-----+
| 192+342 | 23.54+23 |
+-----+-----+
|      534 |      46.54 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Este operador, al igual que el de resta, multiplicación y división, es binario. Como comentamos al hablar de los operadores lógicos, esto no significa que no se puedan asociar, sino que las operaciones se realizan tomando los operandos dos a dos.

## Operador de sustracción o resta

También con la misma lógica, el operador para restar es el -. Otro ejemplo:

```
mysql> SELECT 192-342, 23.54-23;
+-----+-----+
| 192-342 | 23.54-23 |
+-----+-----+
|    -150 |     0.54 |
+-----+-----+
1 row in set (0.02 sec)

mysql>
```

## Operador unitario menos

Este operador, que también usa el símbolo -, se aplica a un único operando, y como resultado se obtiene un valor de signo contrario. Por ejemplo:

```
mysql> SET @x=100;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT -@x;
+-----+
| -@x  |
+-----+
| -100 |
+-----+
1 row in set (0.01 sec)

mysql>
```

## Operador de producto o multiplicación

También es un operador binario, el símbolo usado es el asterisco, \*. Por ejemplo:

```
mysql> SELECT 12343432*3123243, 312*32*12;
+-----+-----+
| 12343432*3123243 | 312*32*12 |
+-----+-----+
```

```
| 38551537589976 | 119808 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

## Operador de cociente o división

El resultado de las divisiones, por regla general, es un número en coma flotante. Por supuesto, también es un operador binario, y el símbolo usado es `/`.

Dividir por cero produce como resultado el valor *NULL*. Por ejemplo:

```
mysql> SELECT 2132143/3123, 4324/25434, 43/0;
+-----+-----+-----+
| 2132143/3123 | 4324/25434 | 43/0 |
+-----+-----+-----+
|          682.72 |          0.17 | NULL |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

## Operador de división entera

Existe otro operador para realizar divisiones, pero que sólo calcula la parte entera del cociente. El operador usado es *DIV*. Por ejemplo:

```
mysql> SELECT 2132143 DIV 3123, 4324 DIV 25434, 43 DIV 0;
+-----+-----+-----+
| 2132143 DIV 3123 | 4324 DIV 25434 | 43 DIV 0 |
+-----+-----+-----+
|          682 |          0 | NULL |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

## Operadores de bits

Todos los operadores de bits trabajan con enteros BIGINT, es decir con 64 bits.

Los operadores son los habituales: o, y, o exclusivo, complemento y rotaciones a derecha e izquierda.

## Operador de bits O

El símbolo empleado es |. Este operador es equivalente al operador OR que vimos para álgebra de Boole, pero se aplica bit a bit entre valores enteros de 64 bits.

Las tablas de verdad para estos operadores son más simples, ya que los bits no pueden tomar valores nulos:

Bit A	Bit B	A   B
0	0	0
0	1	1
1	0	1
1	1	1

Las operaciones con operadores de bits se realizan tomando los bits de cada operador uno a uno.  
Ejemplo:

```

11100011 11001010 010101010 11100011 00001111 10101010 11111111 00000101
O 00101101 11011110 100010100 11101011 11010010 11010101 00101001 11010010
11101111 11011110 110111110 11101011 11011111 11111111 11111111 11010111

```

Por ejemplo:

```

mysql> SELECT 234 | 334, 32 | 23, 15 | 0;
+-----+-----+-----+
| 234 | 334 | 32 | 23 | 15 | 0 |
+-----+-----+-----+
|      494 |      55 |      15 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>

```

## Operador de bits Y

El símbolo empleado es `&`. Este operador es equivalente al operador AND que vimos para álgebra de Boole, pero aplicado bit a bit entre valores enteros de 64 bits.

La tabla de verdad para este operador es:

Bit A	Bit B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Al igual que con el operador `|`, con el operador `&` las operaciones se realizan tomando los bits de cada operador uno a uno. Ejemplo:

```

11100011 11001010 010101010 11100011 00001111 10101010 11111111 00000101
Y 00101101 11011110 100010100 11101011 11010010 11010101 00101001 11010010
00100001 11001000 000000000 11100011 00000010 10000000 00101001 00000000

```

Por ejemplo:

```

mysql> SELECT 234 & 334, 32 & 23, 15 & 0;
+-----+-----+-----+
| 234 & 334 | 32 & 23 | 15 & 0 |
+-----+-----+-----+
|          74 |          0 |          0 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>

```

## Operador de bits O exclusivo

El símbolo empleado es `^`. Este operador es equivalente al operador XOR que vimos para álgebra de Boole, pero aplicado bit a bit entre valores enteros de 64 bits.

La tabla de verdad para este operador es:

Bit A	Bit B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Al igual que con los operadores anteriores, con el operador ^ las operaciones se realizan tomando los bits de cada operador uno a uno. Ejemplo:

```

11100011 11001010 010101010 11100011 00001111 10101010 11111111 00000101
^ 00101101 11011110 100010100 11101011 11010010 11010101 00101001 11010010
11001110 00010100 110111110 00001000 11011101 01111111 11010110 11010111

```

Por ejemplo:

```

mysql> SELECT 234 ^ 334, 32 ^ 23, 15 ^ 0;
+-----+-----+-----+
| 234 ^ 334 | 32 ^ 23 | 15 ^ 0 |
+-----+-----+-----+
|          420 |          55 |          15 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>

```

## Operador de bits de complemento

El símbolo empleado es ~. Este operador es equivalente al operador NOT que vimos para álgebra de Boole, pero aplicado bit a bit entre valores enteros de 64 bits.

Se trata de un operador unitario, y la tabla de verdad es:

Bit A	~A
0	1
1	0

Al igual que con los operadores anteriores, con el operador ~ las operaciones se realizan tomando los

bits del operador uno a uno. Ejemplo:

```
~ 11100011 11001010 010101010 11100011 00001111 10101010 11111111 00000101
   00011100 00110101 101010101 00011100 11110000 01010101 00000000 11111010
```

Por ejemplo:

```
mysql> SELECT ~234, ~32, ~15;
+-----+-----+-----+
| ~234          | ~32          | ~15          |
+-----+-----+-----+
| 18446744073709551381 | 18446744073709551583 | 18446744073709551600 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Como vemos en el ejemplo, el resultado de aplicar el operador de complemento no es un número negativo. Esto es porque si no se especifica lo contrario, se usan valores **BIGINT** sin signo.

Si se fuerza un tipo, el resultado sí será un número de signo contrario. Por ejemplo:

```
mysql> SET @x = ~1;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x;
+-----+
| @x    |
+-----+
| -2    |
+-----+
1 row in set (0.00 sec)

mysql>
```

Para los que no estén familiarizados con el álgebra binaria, diremos que para conseguir el negativo de un número no basta con calcular su complemento. Además hay que sumar al resultado una unidad:

```
mysql> SET @x = ~1 +1, @y = 1;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT @x + @y;
+-----+
| @x + @y |
+-----+
|         0 |
+-----+
1 row in set (0.00 sec)

mysql>
```

## Operador de desplazamiento a la izquierda

El símbolo empleado es `<<`. Se trata de un operador binario. El resultado es que los bits del primer operando se desplazan a la izquierda tantos bits como indique el segundo operando. Por la derecha se introducen otros tantos bits con valor 0. Los bits de la parte izquierda que no caben en los 64 bits, se pierden.

```
11100011 11001010 010101010 11100011 00001111 10101010 11111111 00000101
<< 12
10100101 010101110 00110000 11111010 10101111 11110000 01010000 00000000
```

El resultado, siempre que no se pierdan bits por la izquierda, equivale a multiplicar el primer operando por dos para cada desplazamiento.

Por ejemplo:

```
mysql> SELECT 234 << 25, 32 << 5, 15 << 1;
+-----+-----+-----+
| 234 << 25 | 32 << 5 | 15 << 1 |
+-----+-----+-----+
| 7851737088 |      1024 |      30 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

## Operador de desplazamiento a la derecha

El símbolo empleado es `>>`. Se trata de un operador binario. El resultado es que los bits del primer

operando se desplazan a la derecha tantos bits como indique el segundo operando. Por la izquierda se introducen otros tantos bits con valor 0. Los bits de la parte derecha que no caben en los 64 bits, se pierden.

```
11100011 11001010 010101010 11100011 00001111 10101010 11111111 00000101
>> 7
00000001 11000111 10010100 101010101 11000110 00011111 01010101 11111110
```

El resultado equivale a dividir el primer operando por dos para cada desplazamiento.

Por ejemplo:

```
mysql> SELECT 234 >> 25, 32 >> 5, 15 >> 1;
+-----+-----+-----+
| 234 >> 25 | 32 >> 5 | 15 >> 1 |
+-----+-----+-----+
|          0 |         1 |         7 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

## Contar bits

El último operador de bits del que dispone **MySQL** es *BIT\_COUNT()*. Este operador devuelve el número de bits iguales a 1 que contiene el argumento especificado. Por ejemplo:

```
mysql> SELECT BIT_COUNT(15), BIT_COUNT(12);
+-----+-----+
| BIT_COUNT(15) | BIT_COUNT(12) |
+-----+-----+
|              4 |              2 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

## Operadores de control de flujo

En **MySQL** no siempre es sencillo distinguir los operadores de las funciones. En el caso del control de flujo sólo veremos un operador, el *CASE*. El resto los veremos en el capítulo de funciones.

## Operador *CASE*

Existen dos sintaxis alternativas para *CASE*:

```
CASE valor WHEN [valor1] THEN resultado1 [WHEN [valori] THEN
resultadoi ...] [ELSE resultado] END
CASE WHEN [condición1] THEN resultado1 [WHEN [condicióni] THEN
resultadoi ...] [ELSE resultado] END
```

La primera forma devuelve el resultado para el *valor<sub>i</sub>* que coincida con *valor*.

La segunda forma devuelve el resultado para la primera condición verdadera.

Si no hay coincidencias, se devuelve el valor asociado al *ELSE*, o *NULL* si no hay parte *ELSE*.

```
mysql> SET @x=1;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT CASE @x WHEN 1 THEN "uno"
-> WHEN 2 THEN "varios"
-> ELSE "muchos" END\G
***** 1. row *****
CASE @x WHEN 1 THEN "uno"
WHEN 2 THEN "varios"
ELSE "muchos" END: uno
1 row in set (0.02 sec)

mysql> SET @x=2;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT CASE WHEN @x=1 THEN "uno"
-> WHEN @x=2 THEN "varios"
-> ELSE "muchos" END\G
***** 1. row *****
CASE WHEN @x=1 THEN "uno"
WHEN @x=2 THEN "varios"
ELSE "muchos" END: varios
1 row in set (0.00 sec)
```

```
mysql>
```

## Operadores para cadenas

MySQL dispone de varios operadores para comparación de cadenas, con patrones y con [expresiones regulares](#).

### Operador *LIKE*

El operador *LIKE* se usa para hacer comparaciones entre cadenas y patrones. El resultado es verdadero (1) si la cadena se ajusta al patrón, y falso (0) en caso contrario. Tanto si la cadena como el patrón son *NULL*, el resultado es *NULL*. La sintaxis es:

```
<expresión> LIKE <patrón> [ESCAPE 'carácter_escape']
```

Los patrones son cadenas de caracteres en las que pueden aparecer, en cualquier posición, los caracteres especiales '%' y '\_'. El significado de esos caracteres se puede ver en la tabla siguiente:

Carácter	Descripción
%	Coincidencia con cualquier número de caracteres, incluso ninguno.
_	Coincidencia con un único carácter.

Por ejemplo:

```
mysql> SELECT "hola" LIKE "_o%";
+-----+
| "hola" LIKE "_o%" |
+-----+
|                   1 |
+-----+
1 row in set (0.00 sec)

mysql>
```

La cadena "hola" se ajusta a "\_o%", ya que el carácter 'h' se ajusta a la parte '\_' del patrón, y la subcadena "la" a la parte '%'

La comparación es independiente del tipo de los caracteres, es decir, *LIKE* no distingue mayúsculas de minúsculas, salvo que se indique lo contrario (ver operadores de casting):

```
mysql> SELECT "hola" LIKE "HOLA";
+-----+
| "hola" LIKE "HOLA" |
+-----+
|                1 |
+-----+
1 row in set (0.01 sec)

mysql>
```

Como siempre que se usan caracteres concretos para crear patrones, se presenta la dificultad de hacer comparaciones cuando se deben buscar precisamente esos caracteres concretos. Esta dificultad se suele superar mediante secuencias de escape. Si no se especifica nada en contra, el carácter que se usa para escapar es '\'. De este modo, si queremos que nuestro patrón contenga los caracteres '%' o '\_', los escaparemos de este modo: '\%' y '\\_':

```
mysql> SELECT "%_%" LIKE "_\_\%";
+-----+
| "%_%" LIKE "_\_\%" |
+-----+
|                1 |
+-----+
1 row in set (0.00 sec)

mysql>
```

Pero **MySQL** nos permite usar otros caracteres para crear secuencias de escape, para eso se usa la cláusula opcional *ESCAPE*:

```
mysql> SELECT "%_%" LIKE "_!_!" ESCAPE '!';
+-----+
| "%_%" LIKE "_!_!" ESCAPE '!' |
+-----+
|                1 |
+-----+
1 row in set (0.00 sec)

mysql>
```

En **MySQL**, *LIKE* también funciona con expresiones numéricas. (Esto es una extensión a SQL.)

```
mysql> SELECT 1450 LIKE "1%0";
+-----+
| 1450 LIKE "1%0" |
+-----+
|                1 |
+-----+
1 row in set (0.00 sec)

mysql>
```

El carácter de escape no se aplica sólo a los caracteres '%' y '\_'. **MySQL** usa la misma sintaxis que C para las cadenas, de modo que los caracteres como '\n', '\r', etc también son secuencias de escape, y si se quieren usar como literales, será necesario escaparlos también.

## Operador *NOT LIKE*

La sintaxis es:

```
<expresión> NOT LIKE <patrón> [ESCAPE 'carácter_escape']
```

Equivale a:

```
NOT (<expresión> LIKE <patrón> [ESCAPE 'carácter_escape'])
```

## Operadores *REGEXP* y *RLIKE*

La sintaxis es:

```
<expresión> RLIKE <patrón>
<expresión> REGEXP <patrón>
```

Al igual que *LIKE* el operador *REGEXP* (y su equivalente *RLIKE*), comparan una expresión con un patrón, pero en este caso, el patrón puede ser una expresión regular extendida.

El valor de retorno es verdadero (1) si la expresión coincide con el patrón, en caso contrario devuelve un valor falso (0). Tanto si la expresión como el patrón son nulos, el resultado es *NULL*.

El patrón no tiene que ser necesariamente una cadena, puede ser una expresión o una columna de una tabla.

```
mysql> SELECT 'a' REGEXP '^[a-d]';
+-----+
| 'a' REGEXP '^[a-d]' |
+-----+
|          1          |
+-----+
1 row in set (0.08 sec)

mysql>
```

## Operadores *NOT REGEXP* y *NOT RLIKE*

La sintaxis es:

```
<expresión> NOT RLIKE <patrón>
<expresión> NOT REGEXP <patrón>
```

Que equivalen a:

```
NOT (<expresión> REGEXP <patrón>)
```

## Operadores de casting

En realidad sólo hay un operador de casting: *BINARY*.

### Operador *BINARY*

El operador *BINARY* convierte una cadena de caracteres en una cadena binaria.

Si se aplica a una cadena que forma parte de una comparación, esta se hará de forma sensible al tipo, es decir, se distinguirán mayúsculas de minúsculas.

También hace que los espacios al final de la cadena se tengan en cuenta en la comparación.

```
mysql> SELECT 'a' = 'A', 'a' = BINARY 'A';
+-----+-----+
| 'a' = 'A' | 'a' = BINARY 'A' |
+-----+-----+
|          1 |                0 |
+-----+-----+
1 row in set (0.03 sec)

mysql> SELECT 'a' = 'a ', 'a' = BINARY 'a ';
+-----+-----+
| 'a' = 'a ' | 'a' = BINARY 'a ' |
+-----+-----+
|          1 |                0 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Cuando se usa en comparaciones, *BINARY* afecta a la comparación en conjunto, es indiferente que se aplique a cualquiera de las dos cadenas.

## Tabla de precedencia de operadores

Las precedencias de los operadores son las que se muestran en la siguiente tabla, empezando por la menor:

Operador
:=
, OR, XOR
&&, AND
NOT
BETWEEN, CASE, WHEN, THEN, ELSE
=, <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
&
<<, >>

-, +

\*, /, DIV, %, MOD

^

- (unitario), ~ (complemento)

!

BINARY, COLLATE

## Paréntesis

Como en cualquier otro lenguaje, los paréntesis se pueden usar para forzar el orden de la evaluación de determinadas operaciones dentro de una expresión. Cualquier expresión entre paréntesis adquiere mayor precedencia que el resto de las operaciones en el mismo nivel de paréntesis.

```
mysql> SELECT 10+5*2, (10+5)*2;
```

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

```
| 10+5*2 | (10+5)*2 |
```

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

```
|      20 |       30 |
```

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

```
1 row in set (0.00 sec)
```

```
mysql>
```

# 11 Lenguaje SQL

## Funciones

Si consideramos que **MySQL** es rico en lo que respecta a operadores, en lo que se refiere a funciones, podemos considerarlo millonario. **MySQL** dispone de multitud de funciones.

Pero no las explicaremos aquí, ya que este curso incluye una referencia completa. Tan sólo las agruparemos por tipos, e incluiremos los enlaces correspondientes a la documentación de cada una.

### Funciones de control de flujo

Las funciones de esta categoría son:

- [IF](#) Elección en función de una expresión booleana
- [IFNULL](#) Elección en función de si el valor de una expresión es *NULL*
- [NULLIF](#) Devuelve *NULL* en función del valor de una expresión

### Funciones matemáticas

Las funciones de la categoría de matemáticas son:

- [ABS](#) Devuelve el valor absoluto
- [ACOS](#) Devuelve el arcocoseno
- [ASIN](#) Devuelve el arcoseno
- [ATAN y ATAN2](#) Devuelven el arcotangente
- [CEILING y CEIL](#) Redondeo hacia arriba
- [COS](#) Coseno de un ángulo
- [COT](#) Cotangente de un ángulo
- [CRC32](#) Cálculo de comprobación de redundancia cíclica
- [DEGREES](#) Conversión de grados a radianes
- [EXP](#) Cálculo de potencias de *e*
- [FLOOR](#) Redondeo hacia abajo
- [LN](#) Logaritmo natural

<u>LOG</u>	Logaritmo en base arbitraria
<u>LOG10</u>	Logaritmo en base 10
<u>LOG2</u>	Logaritmo en base dos
<u>MOD o %</u>	Resto de una división entera
<u>PI</u>	Valor del número $\pi$
<u>POW o POWER</u>	Valor de potencias
<u>RADIANS</u>	Conversión de radianes a grados
<u>RAND</u>	Valores aleatorios
<u>ROUND</u>	Cálculo de redondeos
<u>SIGN</u>	Devuelve el signo
<u>SIN</u>	Cálculo del seno de un ángulo
<u>SQRT</u>	Cálculo de la raíz cuadrada
<u>TAN</u>	Cálculo de la tangente de un ángulo
<u>TRUNCATE</u>	Elimina decimales

## Funciones de cadenas

Las funciones para tratamiento de cadenas de caracteres son:

<u>ASCII</u>	Valor de código ASCII de un carácter
<u>BIN</u>	Conversión a binario
<u>BIT_LENGTH</u>	Cálculo de longitud de cadena en bits
<u>CHAR</u>	Convierte de ASCII a carácter
<u>CHAR_LENGTH o CHARACTER_LENGTH</u>	Cálculo de longitud de cadena en caracteres
<u>COMPRESS</u>	Comprime una cadena de caracteres
<u>CONCAT</u>	Concatena dos cadenas de caracteres
<u>CONCAT_WS</u>	Concatena cadenas con separadores
<u>CONV</u>	Convierte números entre distintas bases
<u>ELT</u>	Elección entre varias cadenas
<u>EXPORT_SET</u>	Expresiones binarias como conjuntos
<u>FIELD</u>	Busca el índice en listas de cadenas
<u>FIND_IN_SET</u>	Búsqueda en listas de cadenas

<u>HEX</u>	Conversión de números a hexadecimal
<u>INSERT</u>	Inserta una cadena en otra
<u>INSTR</u>	Busca una cadena en otra
<u>LEFT</u>	Extraer parte izquierda de una cadena
<u>LENGTH</u> u <u>OCTET_LENGTH</u>	Calcula la longitud de una cadena en bytes
<u>LOAD_FILE</u>	Lee un fichero en una cadena
<u>LOCATE</u> o <u>POSITION</u>	Encontrar la posición de una cadena dentro de otra
<u>LOWER</u> o <u>LCASE</u>	Convierte una cadena a minúsculas
<u>LPAD</u>	Añade caracteres a la izquierda de una cadena
<u>LTRIM</u>	Elimina espacios a la izquierda de una cadena
<u>MAKE_SET</u>	Crea un conjunto a partir de una expresión binaria
<u>OCT</u>	Convierte un número a octal
<u>ORD</u>	Obtiene el código ASCII, incluso con caracteres multibyte
<u>QUOTE</u>	Entrecomilla una cadena
<u>REPEAT</u>	Construye una cadena como una repetición de otra
<u>REPLACE</u>	Busca una secuencia en una cadena y la sustituye por otra
<u>REVERSE</u>	Invierte el orden de los caracteres de una cadena
<u>RIGHT</u>	Devuelve la parte derecha de una cadena
<u>RPAD</u>	Inserta caracteres al final de una cadena
<u>RTRIM</u>	Elimina caracteres blancos a la derecha de una cadena
<u>SOUNDEX</u>	Devuelve la cadena "soundex" para una cadena concreta
<u>SOUNDS LIKE</u>	Compara cadenas según su pronunciación
<u>SPACE</u>	Devuelve cadenas consistentes en espacios
<u>SUBSTRING</u> o <u>MID</u>	Extraer subcadenas de una cadena
<u>SUBSTRING_INDEX</u>	Extraer subcadenas en función de delimitadores
<u>TRIM</u>	Elimina sufijos y/o prefijos de una cadena.
<u>UCASE</u> o <u>UPPER</u>	Convierte una cadena a mayúsculas
<u>UNCOMPRESS</u>	Descomprime una cadena comprimida mediante <u>COMPRESS</u>

## UNCOMPRESSED\_LENGTH

Calcula la longitud original de una cadena comprimida

## UNHEX

Convierte una cadena que representa un número hexadecimal a cadena de caracteres

## Funciones de comparación de cadenas

Además de los operadores que vimos para la comparación de cadenas, existe una función:

STRCMP Compara cadenas

## Funciones de fecha

Funciones para trabajar con fechas:

### ADDDATE

Suma un intervalo de tiempo a una fecha

### ADDTIME

Suma tiempos

### CONVERT\_TZ

Convierte tiempos entre distintas zonas horarias

### CURDATE o CURRENTDATE

Obtener la fecha actual

### CURTIME o CURRENT\_TIME

Obtener la hora actual

### DATE

Extraer la parte correspondiente a la fecha

### DATEDIFF

Calcula la diferencia en días entre dos fechas

### DATE\_ADD

Aritmética de fechas, suma un intervalo de tiempo

### DATE\_SUB

Aritmética de fechas, resta un intervalo de tiempo

### DATE\_FORMAT

Formatea el valor de una fecha

### DAY o DAYOFMONTH

Obtiene el día del mes a partir de una fecha

### DAYNAME

Devuelve el nombre del día de la semana

### DAYOFWEEK

Devuelve el índice del día de la semana

DAYOFYEAR

Devuelve el día del año para una fecha

EXTRACT

Extrae parte de una fecha

FROM\_DAYS

Obtener una fecha a partir de un número de días

FROM\_UNIXTIME

Representación de fechas UNIX en formato de cadena

GET\_FORMAT

Devuelve una cadena de formato

HOUR

Extrae la hora de un valor time

LAST\_DAY

Devuelve la fecha para el último día del mes de una fecha

MAKEDATE

Calcula una fecha a partir de un año y un día del año

MAKETIME

Calcula un valor de tiempo a partir de una hora, minuto y segundo

MICROSECOND

Extrae los microsegundos de una expresión de fecha/hora o de hora

MINUTE

Extrae el valor de minutos de una expresión time

MONTH

Devuelve el mes de una fecha

MONTHNAME

Devuelve el nombre de un mes para una fecha

NOW o CURRENT\_TIMESTAMP o LOCALTIME o LOCALTIMESTAMP o SYSDATE

Devuelve la fecha y hora actual

PERIOD\_ADD

Añade meses a un periodo (año/mes)

PERIOD\_DIFF

Calcula la diferencia de meses entre dos periodos (año/mes)

QUARTER

Devuelve el cuarto del año para una fecha

SECOND

Extrae el valor de segundos de una expresión time

SEC\_TO\_TIME

Convierte una cantidad de segundos a horas, minutos y segundos

STR\_TO\_DATE

Obtiene un valor DATETIME a partir de una cadena con una fecha y una cadena de formato

SUBDATE

Resta un intervalo de tiempo de una fecha

SUBTIME

Resta dos expresiones time

TIME

Extrae la parte de la hora de una expresión fecha/hora

TIMEDIFF

Devuelve en tiempo entre dos expresiones de tiempo

TIMESTAMP

Convierte una expresión de fecha en fecha/hora o suma un tiempo a una fecha

TIMESTAMPADD

Suma un intervalo de tiempo a una expresión de fecha/hora

TIMESTAMPDIFF

Devuelve la diferencia entre dos expresiones de fecha/hora

TIME\_FORMAT

Formatea un tiempo

TIME\_TO\_SEC

Convierte un tiempo a segundos

TO\_DAYS

Calcula el número de días desde el año cero

UNIX\_TIMESTAMP

Devuelve un timestamp o una fecha en formato UNIX, segundos desde 1070

UTC\_DATE

Devuelve la fecha UTC actual

UTC\_TIME

Devuelve la hora UTC actual

UTC\_TIMESTAMP

Devuelve la fecha y hora UTC actual

WEEK

Calcula el número de semana para una fecha

WEEKDAY

Devuelve el número de día de la semana para una fecha

WEEKOFYEAR

Devuelve el número de la semana del año para una fecha

YEAR

Extrae el año de una fecha

YEARWEEK

Devuelve el año y semana de una fecha

**De búsqueda de texto**

Función de búsqueda de texto:

MATCH**Funciones de casting (conversión de tipos)**CAST o CONVERT Conversión de tipos explícita**Funciones de encriptado**

Funciones de encriptado de datos y de checksum:

AES\_ENCRYPT y AES\_DECRYPT Encriptar y desencriptar datos usando el algoritmo oficial AESDECODE Desencripta una cadena usando una contraseñaENCODE Encripta una cadena usando una contraseñaDES\_DECRYPT Desencripta usando el algoritmo Triple-DESDES\_ENCRYPT Encripta usando el algoritmo Triple-DESENCRYPT Encripta str usando la llamada del sistema Unix *crypt()*MD5 Calcula un checksum MD5 de 128 bits para la cadena stringPASSWORD u OLD\_PASSWORD Calcula una cadena contraseña a partir de la cadena en texto planoSHA o SHA1 Calcula un checksum SHA1 de 160 bits para una cadena**Funciones de información**

Información sobre el sistema:

BENCHMARK Ejecuta una expresión varias vecesCHARSET Devuelve el conjunto de caracteres de una cadenaCOERCIBILITY Devuelve el valor de restricción de colección de una cadenaCOLLATION Devuelve la colección para el conjunto de caracteres de una cadenaCONNECTION\_ID Devuelve el ID de una conexiónCURRENT\_USER Devuelve el nombre de usuario y el del host para la conexión actual

<u><a href="#">DATABASE</a></u>	Devuelve el nombre de la base de datos actual
<u><a href="#">FOUND_ROWS</a></u>	Calcular cuántas filas se hubiesen obtenido en una sentencia SELECT sin la cláusula LIMIT
<u><a href="#">LAST_INSERT_ID</a></u>	Devuelve el último valor generado automáticamente para una columna AUTO_INCREMENT
<u><a href="#">USER</a></u> o <u><a href="#">SESSION_USER</a></u> o <u><a href="#">SYSTEM_USER</a></u>	Devuelve el nombre de usuario y host actual de MySQL
<u><a href="#">VERSION</a></u>	Devuelve la versión del servidor MySQL

## Miscelanea

Funciones generales:

<u><a href="#">DEFAULT</a></u>	Devuelve el valor por defecto para una columna
<u><a href="#">FORMAT</a></u>	Formatea el número según la plantilla '#,###,###.##'
<u><a href="#">GET_LOCK</a></u>	Intenta obtener un bloqueo con el nombre dado
<u><a href="#">INET_ATON</a></u>	Obtiene el entero equivalente a la dirección de red dada en formato de cuarteto con puntos
<u><a href="#">INET_NTOA</a></u>	Obtiene la dirección en formato de cuarteto con puntos dado un entero
<u><a href="#">IS_FREE_LOCK</a></u>	Verifica si un nombre de bloqueo está libre
<u><a href="#">IS_USED_LOCK</a></u>	Verifica si un nombre de bloqueo está en uso
<u><a href="#">MASTER_POS_WAIT</a></u>	Espera hasta que el esclavo alcanza la posición especificada en el diario maestro
<u><a href="#">RELEASE_LOCK</a></u>	Libera un bloqueo
<u><a href="#">UUID</a></u>	Devuelve un identificador único universal

## De grupos

Funciones de grupos:

<u><a href="#">AVG</a></u>	Devuelve el valor medio
<u><a href="#">BIT_AND</a></u>	Devuelve la operación de bits AND para todos los bits de una expresión
<u><a href="#">BIT_OR</a></u>	Devuelve la operación de bits OR para todos los bits de una expresión
<u><a href="#">BIT_XOR</a></u>	Devuelve la operación de bits XOR para todos los bits de una expresión

<u>COUNT</u>	Devuelve el número de valores distintos de NULL en las filas recuperadas por una sentencia SELECT
<u>COUNT DISTINCT</u>	Devuelve el número de valores diferentes, distintos de NULL
<u>GROUP_CONCAT</u>	Devuelve una cadena con la concatenación de los valores de un grupo
<u>MIN</u>	Devuelve el valor mínimo de una expresión
<u>MAX</u>	Devuelve el valor máximo de una expresión
<u>STD o STDDEV</u>	Devuelve la desviación estándar de una expresión
<u>SUM</u>	Devuelve la suma de una expresión
<u>VARIANCE</u>	Devuelve la varianza estándar de una expresión

# 12 Lenguaje SQL

## Consultas multitable

Hasta ahora todas las consultas que hemos usado se refieren sólo a una tabla, pero también es posible hacer consultas usando varias tablas en la misma sentencia [SELECT](#).

Esto nos permite realizar otras dos operaciones de álgebra relacional que aún no hemos visto: [el producto cartesiano](#) y la [composición](#).

### Producto cartesiano

Usaremos el ejemplo de las tablas de *personas2* y *telefonos2* del [capítulo 7](#), e insertaremos algunos datos:

```
mysql> INSERT INTO personas2 (nombre, fecha) VALUES
-> ("Fulanito", "1956-12-14"),
-> ("Menganito", "1975-10-15"),
-> ("Tulanita", "1985-03-17"),
-> ("Fusganita", "1976-08-25");
Query OK, 4 rows affected (0.09 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT * FROM personas2;
+----+-----+-----+
| id | nombre   | fecha       |
+----+-----+-----+
|  1 | Fulanito | 1956-12-14 |
|  2 | Menganito | 1975-10-15 |
|  3 | Tulanita | 1985-03-17 |
|  4 | Fusganita | 1976-08-25 |
+----+-----+-----+
4 rows in set (0.00 sec)
```

```
mysql>
```

Ahora insertaremos datos en la tabla de *telefonos2*:

```
mysql> INSERT INTO telefonos2 (id, numero) VALUES
-> (1, "123456789"),
```

```
-> (1, "145654854"),
-> (1, "152452545"),
-> (2, "254254254"),
-> (4, "456545654"),
-> (4, "441415414");
```

Query OK, 6 rows affected (0.06 sec)

Records: 6 Duplicates: 0 Warnings: 0

```
mysql> SELECT * FROM telefonos2;
```

```
+-----+-----+
| numero | persona |
+-----+-----+
| 123456789 | 1 |
| 145654854 | 1 |
| 152452545 | 1 |
| 254254254 | 2 |
| 456545654 | 4 |
| 441415414 | 4 |
+-----+-----+
```

6 rows in set (0.00 sec)

```
mysql>
```

El producto cartesiano de dos tablas son todas las combinaciones de todas las filas de las dos tablas. Usando una sentencia **SELECT** se hace proyectando todos los atributos de ambas tablas. Los nombres de las tablas se indican en la cláusula *FROM* separados con comas:

```
mysql> SELECT * FROM personas2,telefonos2;
```

```
+-----+-----+-----+-----+-----+
| id | nombre | fecha | numero | id |
+-----+-----+-----+-----+-----+
| 1 | Fulanito | 1956-12-14 | 123456789 | 1 |
| 2 | Menganito | 1975-10-15 | 123456789 | 1 |
| 3 | Tulanita | 1985-03-17 | 123456789 | 1 |
| 4 | Fusganita | 1976-08-25 | 123456789 | 1 |
| 1 | Fulanito | 1956-12-14 | 145654854 | 1 |
| 2 | Menganito | 1975-10-15 | 145654854 | 1 |
| 3 | Tulanita | 1985-03-17 | 145654854 | 1 |
| 4 | Fusganita | 1976-08-25 | 145654854 | 1 |
| 1 | Fulanito | 1956-12-14 | 152452545 | 1 |
| 2 | Menganito | 1975-10-15 | 152452545 | 1 |
| 3 | Tulanita | 1985-03-17 | 152452545 | 1 |
| 4 | Fusganita | 1976-08-25 | 152452545 | 1 |
| 1 | Fulanito | 1956-12-14 | 254254254 | 2 |
| 2 | Menganito | 1975-10-15 | 254254254 | 2 |
```

```

| 3 | Tulanita | 1985-03-17 | 254254254 | 2 |
| 4 | Fusganita | 1976-08-25 | 254254254 | 2 |
| 1 | Fulanito | 1956-12-14 | 456545654 | 4 |
| 2 | Menganito | 1975-10-15 | 456545654 | 4 |
| 3 | Tulanita | 1985-03-17 | 456545654 | 4 |
| 4 | Fusganita | 1976-08-25 | 456545654 | 4 |
| 1 | Fulanito | 1956-12-14 | 441415414 | 4 |
| 2 | Menganito | 1975-10-15 | 441415414 | 4 |
| 3 | Tulanita | 1985-03-17 | 441415414 | 4 |
| 4 | Fusganita | 1976-08-25 | 441415414 | 4 |
+-----+-----+-----+-----+-----+
24 rows in set (0.73 sec)

mysql>

```

Como se ve, la salida consiste en todas las combinaciones de todas las tuplas de ambas tablas.

## Composición (Join)

Recordemos que se trata de un producto cartesiano restringido, las tuplas que se emparejan deben cumplir una determinada condición.

En el álgebra relacional sólo hemos hablado de composiciones en general. Sin embargo, en SQL se trabaja con varios tipos de composiciones.

## Composiciones internas

Todas las composiciones que hemos visto hasta ahora se denominan *composiciones internas*. Para hacer una composición interna se parte de un producto cartesiano y se eliminan aquellas tuplas que no cumplen la condición de la composición.

En el ejemplo anterior tenemos 24 tuplas procedentes del producto cartesiano de las tablas *personas2* y *teléfonos2*. Si la condición para la composición es que *personas2.id=teléfonos2.id*, tendremos que eliminar todas las tuplas en que la condición no se cumpla.

Estas composiciones se denominan internas porque en la salida no aparece ninguna tupla que no esté presente en el producto cartesiano, es decir, la composición se hace en el *interior* del producto cartesiano de las tablas.

Para consultar la sintaxis de las composiciones ver [JOIN](#).

Las composiciones internas usan estas sintaxis:

```
referencia_tabla, referencia_tabla
referencia_tabla [INNER | CROSS] JOIN referencia_tabla [condición]
```

La condición puede ser:

```
ON expresión_condicional | USING (lista_columnas)
```

La coma y *JOIN* son equivalentes, y las palabras *INNER* y *CROSS* son opcionales.

La condición en la cláusula *ON* puede ser cualquier expresión válida para una cláusula *WHERE*, de hecho, en la mayoría de los casos, son equivalentes.

La cláusula *USING* nos permite usar una lista de atributos que deben ser iguales en las dos tablas a componer.

Siguiendo con el mismo ejemplo, la condición más lógica para la composición interna entre *personas2* y *teléfonos2* es la igualdad entre el identificador de persona en la primera tabla y el atributo persona en la segunda:

```
mysql> SELECT * FROM personas2, telefonos2
-> WHERE personas2.id=telefonos2.id;
+----+-----+-----+-----+----+
| id | nombre   | fecha       | numero   | id |
+----+-----+-----+-----+----+
| 1  | Fulanito | 1956-12-14 | 123456789 | 1  |
| 1  | Fulanito | 1956-12-14 | 145654854 | 1  |
| 1  | Fulanito | 1956-12-14 | 152452545 | 1  |
| 2  | Menganito | 1975-10-15 | 254254254 | 2  |
| 4  | Fusganita | 1976-08-25 | 456545654 | 4  |
| 4  | Fusganita | 1976-08-25 | 441415414 | 4  |
+----+-----+-----+-----+----+
6 rows in set (0.73 sec)

mysql>
```

Esta consulta es equivalente a estas otras:

```
mysql> SELECT * FROM personas2 JOIN telefonos2
-> ON (personas2.id = telefonos2.id);
mysql> SELECT * FROM personas2 JOIN telefonos2
-> WHERE (personas2.id = telefonos2.id);
mysql> SELECT * FROM personas2 INNER JOIN telefonos2
-> ON (personas2.id = telefonos2.id);
mysql> SELECT * FROM personas2 CROSS JOIN telefonos2
-> ON (personas2.id = telefonos2.id);
mysql> SELECT * FROM personas2 JOIN telefonos2 USING(id);
```

En cualquier caso, la salida sólo contiene las tuplas que emparejan a personas con sus números de teléfono. Las tuplas correspondientes a personas que no tienen ningún número no aparecen, como por ejemplo las correspondientes a "Tulanita". Para las personas con varios números, se repiten los datos de la persona para cada número, por ejemplo con "Fulanito" o "Fusganita".

## Composición interna natural

Consiste en una proyección sobre un producto cartesiano restringido. Es decir, sólo elegimos determinadas columnas de ambas tablas, en lugar de seleccionar todas.

Podemos hacer esto a partir de una composición general, eligiendo todas las columnas menos las repetidas:

```
mysql> SELECT personas2.id,nombre,fecha,numero
-> FROM personas2, telefonos2
-> WHERE personas2.id=telefonos2.id;
+----+-----+-----+-----+
| id | nombre   | fecha   | numero |
+----+-----+-----+-----+
| 1  | Fulanito | 1956-12-14 | 123456789 |
| 1  | Fulanito | 1956-12-14 | 145654854 |
| 1  | Fulanito | 1956-12-14 | 152452545 |
| 2  | Menganito | 1975-10-15 | 254254254 |
| 4  | Fusganita | 1976-08-25 | 456545654 |
| 4  | Fusganita | 1976-08-25 | 441415414 |
+----+-----+-----+-----+
6 rows in set (0.00 sec)

mysql>
```

Como la columna *id* existe en ambas tablas estamos obligados a usar el nombre completo para esta columna. En este caso hemos optado por *personas2.id*, pero hubiese sido igual usar *telefonos2.id*.

También podemos definir alias para las tablas, y conseguir una consulta más compacta:

```
mysql> SELECT t1.id,nombre,fecha,numero
-> FROM personas2 AS t1, telefonos2 AS t2
-> WHERE t1.id=t2.id;
```

Por supuesto, podemos usar *JOIN* y *ON* en lugar de la coma y *WHERE*:

```
mysql> SELECT t1.id,nombre,fecha,numero
-> FROM personas2 AS t1 JOIN telefonos2 AS t2
-> ON t1.id=t2.id;
```

Pero tenemos una sintaxis alternativa mucho mejor para hacer composiciones internas naturales:

```
referencia_tabla NATURAL JOIN referencia_tabla
```

Por ejemplo:

```
mysql> SELECT * FROM personas2 NATURAL JOIN telefonos2;
+----+-----+-----+-----+
| id | nombre   | fecha   | numero |
+----+-----+-----+-----+
| 1  | Fulanito | 1956-12-14 | 123456789 |
| 1  | Fulanito | 1956-12-14 | 145654854 |
| 1  | Fulanito | 1956-12-14 | 152452545 |
| 2  | Menganito | 1975-10-15 | 254254254 |
| 4  | Fusganita | 1976-08-25 | 456545654 |
| 4  | Fusganita | 1976-08-25 | 441415414 |
+----+-----+-----+-----+
6 rows in set (0.02 sec)

mysql>
```

## Composiciones externas

Al contrario que con las composiciones internas, las externas no proceden de un producto cartesiano. Por lo tanto, en estas pueden aparecer tuplas que no aparecen en el producto cartesiano.

Para hacer una composición externa se toman las tuplas de una de las tablas una a una y se combinan con las tuplas de la otra.

Como norma general se usa un índice para localizar las tuplas de la segunda tabla que cumplen la condición, y para cada tupla encontrada se añade una fila a la tabla de salida.

Si no existe ninguna tupla en la segunda tabla que cumpla las condiciones, se combina la tupla de la primera con una nula de la segunda.

En nuestro ejemplo se tomaría la primera tupla de *personas2*, con un valor de *id* igual a 1, y se busca en la tabla *telefonos2* las tuplas con un valor de *id* igual a 1. Lo mismo para la segunda tupla, con *id* igual a 2.

En la tercera el *id* es 3, y no existe ninguna tupla en *telefonos2* con un valor de *id* igual a 3, por lo tanto se combina la tupla de *personas2* con una tupla de *telefonos2* con todos los atributos igual a *NULL*.

Por ejemplo:

```
mysql> SELECT * FROM personas2 LEFT JOIN telefonos2 USING(id);
```

id	nombre	fecha	numero	id
1	Fulanito	1956-12-14	123456789	1
1	Fulanito	1956-12-14	145654854	1
1	Fulanito	1956-12-14	152452545	1
2	Menganito	1975-10-15	254254254	2
<b>3</b>	<b>Tulanita</b>	<b>1985-03-17</b>	<b>NULL</b>	<b>NULL</b>
4	Fusganita	1976-08-25	456545654	4
4	Fusganita	1976-08-25	441415414	4

(1)

7 rows in set (0.05 sec)

```
mysql>
```

La quinta fila (1), tiene valores *NULL* para *numero* e *id* de *telefonos2*, ya que no existen tuplas en esa tabla con un valor de *id* igual a 3.

Las sintaxis para composiciones externas son:

```
referencia_tabla LEFT [OUTER] JOIN referencia_tabla [join_condition]
referencia_tabla NATURAL LEFT [OUTER] JOIN referencia_tabla
```

```
referencia_tabla RIGHT [OUTER] JOIN referencia_tabla [condición]
referencia_tabla NATURAL RIGHT [OUTER] JOIN referencia_tabla
```

La condición puede ser:

```
ON expresión_condicional | USING (lista_columnas)
```

La palabra *OUTER* es opcional.

Existen dos grupos de composiciones externas: izquierda y derecha, dependiendo de cual de las tablas se lea en primer lugar.

## Composición externa izquierda

En estas composiciones se recorre la tabla de la izquierda y se buscan tuplas en la de la derecha. Se crean usando la palabra *LEFT* (izquierda, en inglés).

Las sintaxis para la composición externa izquierda es:

```
referencia_tabla LEFT [OUTER] JOIN referencia_tabla [condición]
```

Veamos un ejemplo. Para empezar, crearemos un par de tablas:

```
mysql> CREATE TABLE tabla1 (
  -> id INT NOT NULL,
  -> nombre CHAR(10),
  -> PRIMARY KEY (id));
Query OK, 0 rows affected (0.42 sec)

mysql> CREATE TABLE tabla2 (
  -> id INT NOT NULL,
  -> numero INT,
  -> PRIMARY KEY (id));
Query OK, 0 rows affected (0.11 sec)

mysql>
```

E insertaremos algunos datos:

```
mysql> INSERT INTO tabla1 VALUES
-> (5, "Juan"),
-> (6, "Pedro"),
-> (7, "José"),
-> (8, "Fernando");
Query OK, 4 rows affected (0.06 sec)
Records: 4 Duplicates: 0 Warnings: 0

mysql> INSERT INTO tabla2 VALUES
-> (3, 30),
-> (4, 40),
-> (5, 50),
-> (6, 60);
Query OK, 5 rows affected (0.05 sec)
Records: 5 Duplicates: 0 Warnings: 0

mysql>
```

La composición izquierda sería:

```
mysql> SELECT * FROM tabla1 LEFT JOIN tabla2 USING(id);
+----+-----+-----+-----+
| id | nombre  | id  | numero |
+----+-----+-----+-----+
| 5  | Juan    | 5   | 50     |
| 6  | Pedro   | 6   | 60     |
| 7  | José    | NULL | NULL   |
| 8  | Fernando | NULL | NULL   |
+----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

Se puede ver que aparecen dos filas con valores *NULL*, para los *id* 7 y 8.

En contraposición, una composición interna dará esta salida:

```
mysql> SELECT * FROM tabla1 JOIN tabla2 USING(id);
+----+-----+-----+-----+
| id | nombre | id  | numero |
+----+-----+-----+-----+
| 5  | Juan   | 5   | 50     |
```

```
| 6 | Pedro | 6 | 60 |
+---+-----+---+-----+
2 rows in set (0.06 sec)

mysql>
```

## Composición externa derecha

En este caso se recorre la tabla de la derecha y se buscan tuplas que cumplan la condición en la tabla izquierda.

La sintaxis es equivalente:

```
referencia_tabla LEFT [OUTER] JOIN referencia_tabla [condición]
```

Usando las mismas tablas que en el ejemplo anterior:

```
mysql> SELECT * FROM tabla1 RIGHT JOIN tabla2 USING(id);
+-----+-----+-----+-----+
| id    | nombre | id    | numero |
+-----+-----+-----+-----+
| NULL  | NULL   | 3     | 30     |
| NULL  | NULL   | 4     | 40     |
| 5     | Juan   | 5     | 50     |
| 6     | Pedro  | 6     | 60     |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

Es lo mismo usar una composición derecha de las tablas tabla1 y tabla2 que una composición izquierda de las tablas tabla2 y tabla1. Es decir, la consulta anterior es equivalente a esta otra:

```
mysql> SELECT * FROM tabla2 LEFT JOIN tabla1 USING(id);
```

## Composiciones naturales externas

Por supuesto, también podemos hacer composiciones externas naturales:

```
referencia_tabla NATURAL LEFT [OUTER] JOIN referencia_tabla
referencia_tabla NATURAL RIGHT [OUTER] JOIN referencia_tabla
```

El problema es que si existen tuplas añadidas con respecto a la composición interna, no se eliminará ninguna columna. Los mismos ejemplos anteriores, como composiciones naturales externas serían:

```
mysql> SELECT * FROM tabla1 NATURAL LEFT JOIN tabla2;
```

id	nombre	id	numero
5	Juan	5	50
6	Pedro	6	60
7	José	NULL	NULL
8	Fernando	NULL	NULL

```
4 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM tabla1 NATURAL RIGHT JOIN tabla2;
```

id	nombre	id	numero
NULL	NULL	3	30
NULL	NULL	4	40
5	Juan	5	50
6	Pedro	6	60

```
4 rows in set (0.00 sec)
```

```
mysql>
```

## Uniones

También es posible realizar la operación de álgebra relacional [unión](#) entre varias tablas o [proyecciones](#) de tablas.

Para hacerlo se usa la sentencia [UNION](#) que permite combinar varias sentencias [SELECT](#) para crear una única tabla de salida.

Las condiciones para que se pueda crear una unión son las mismas que vimos al estudiar el álgebra relacional: las relaciones a unir deben tener el mismo número de atributos, y además deben ser de dominios compatibles.

Veamos un ejemplo:

```
mysql> CREATE TABLE stock1 (  
-> id INT NOT NULL,  
-> nombre VARCHAR(30),  
-> cantidad INT,  
-> PRIMARY KEY (id));  
Query OK, 0 rows affected (0.08 sec)  
  
mysql> CREATE TABLE stock2 (  
-> id INT NOT NULL,  
-> nombre VARCHAR(40),  
-> cantidad SMALLINT,  
-> PRIMARY KEY (id));  
Query OK, 0 rows affected (0.16 sec)  
  
mysql> CREATE TABLE stock3 (  
-> id INT NOT NULL,  
-> nombre VARCHAR(35),  
-> numero MEDIUMINT,  
-> PRIMARY KEY (id));  
Query OK, 0 rows affected (0.08 sec)  
  
mysql> INSERT INTO stock1 VALUES  
-> (1, "tornillo M3x12", 100),  
-> (2, "tornillo M3x15", 120),  
-> (3, "tornillo M4x25", 120),  
-> (4, "tornillo M5x30", 200);  
Query OK, 4 rows affected (0.03 sec)  
Records: 4 Duplicates: 0 Warnings: 0  
  
mysql> INSERT INTO stock2 VALUES  
-> (10, "tuerca M4", 120),  
-> (11, "tuerca M3", 100),  
-> (12, "tuerca M5", 87);  
Query OK, 3 rows affected (0.05 sec)  
Records: 3 Duplicates: 0 Warnings: 0  
  
mysql> INSERT INTO stock3 VALUES  
-> (20, "varilla 10", 23),  
-> (1, "tornillo M3x12", 22),  
-> (21, "varilla 12", 32),  
-> (11, "tuerca M3", 22);  
Query OK, 4 rows affected (0.03 sec)  
Records: 4 Duplicates: 0 Warnings: 0
```

```
mysql>
```

Podemos crear una unión de las tres tablas, a pesar de que los nombres y tamaños de algunas columnas sean diferentes:

```
mysql> SELECT * FROM stock1 UNION
-> SELECT * FROM stock2 UNION
-> SELECT * FROM stock3;
```

id	nombre	cantidad
1	tornillo M3x12	100
2	tornillo M3x15	120
3	tornillo M4x25	120
4	tornillo M5x30	200
10	tuerca M4	120
11	tuerca M3	100
12	tuerca M5	87
1	tornillo M3x12	22
11	tuerca M3	22
20	varilla 10	23
21	varilla 12	32

```
11 rows in set (0.00 sec)

mysql>
```

El resultado se puede ordenar usando *ORDER BY* y también podemos seleccionar un número limitado de filas mediante *LIMIT*:

```
mysql> (SELECT * FROM stock1) UNION
-> (SELECT * FROM stock2) UNION
-> (SELECT * FROM stock3) ORDER BY id LIMIT 6;
```

id	nombre	cantidad
1	tornillo M3x12	100
1	tornillo M3x12	22
2	tornillo M3x15	120
3	tornillo M4x25	120
4	tornillo M5x30	200
10	tuerca M4	120

```
6 rows in set (0.00 sec)
```

```
mysql>
```

Dentro de cada sentencia SELECT se aplican todas las cláusulas, proyecciones y selecciones que se quiera, como en cualquier SELECT normal.

La sintaxis completa incluye dos modificadores:

```
SELECT ...
UNION [ALL | DISTINCT]
SELECT ...
  [UNION [ALL | DISTINCT]
   SELECT ...]
```

Los modificadores *ALL* y *DISTINCT* son opcionales, y si no se usa ninguno el comportamiento es el mismo que si se usa *DISTINCT*.

Con *ALL* se muestran todas las filas, aunque estén repetidas, con *DISTINCT* sólo se muestra una copia de cada fila:

```
mysql> SELECT id,nombre FROM stock1 UNION
-> SELECT id,nombre FROM stock2 UNION
-> SELECT id,nombre FROM stock3;
```

```
+----+-----+
| id | nombre          |
+----+-----+
|  1 | tornillo M3x12  |
|  2 | tornillo M3x15  |
|  3 | tornillo M4x25  |
|  4 | tornillo M5x30  |
| 10 | tuerca M4       |
| 11 | tuerca M3       |
| 12 | tuerca M5       |
| 20 | varilla 10      |
| 21 | varilla 12      |
+----+-----+
```

```
9 rows in set (0.00 sec)
```

```
mysql> SELECT id,nombre FROM stock1 UNION ALL
-> SELECT id,nombre FROM stock2 UNION ALL
-> SELECT id,nombre FROM stock3;
```

```
+-----+-----+
| id | nombre |
+-----+-----+
| 1 | tornillo M3x12 |
| 2 | tornillo M3x15 |
| 3 | tornillo M4x25 |
| 4 | tornillo M5x30 |
| 10 | tuerca M4 |
| 11 | tuerca M3 |
| 12 | tuerca M5 |
| 1 | tornillo M3x12 |
| 11 | tuerca M3 |
| 20 | varilla 10 |
| 21 | varilla 12 |
+-----+-----+
11 rows in set (0.00 sec)

mysql>
```

Sobre el resultado final no se pueden aplicar otras cláusulas como *GROUP BY*.

# 13 Lenguaje SQL

## Usuarios y privilegios

Hasta ahora hemos usado sólo el usuario 'root', que es el administrador, y que dispone de todos los privilegios disponibles en **MySQL**.

Sin embargo, normalmente no será una buena práctica dejar que todos los usuario con acceso al servidor tengan todos los privilegios. Para conservar la integridad de los datos y de las estructuras será conveniente que sólo algunos usuarios puedan realizar determinadas tareas, y que otras, que requieren mayor conocimiento sobre las estructuras de bases de datos y tablas, sólo puedan realizarse por un número limitado y controlado de usuarios.

Los conceptos de usuarios y privilegios están íntimamente relacionados. No se pueden crear usuarios sin asignarle al mismo tiempo privilegios. De hecho, la necesidad de crear usuarios está ligada a la necesidad de limitar las acciones que tales usuarios pueden llevar a caobo.

**MySQL** permite definir diferentes usuarios, y además, asignar a cada uno determinados privilegios en distintos niveles o categorías de ellos.

### Niveles de privilegios

En **MySQL** existen cinco niveles distintos de privilegios:

**Globales:** se aplican al conjunto de todas las bases de datos en un servidor. Es el nivel más alto de privilegio, en el sentido de que su ámbito es el más general.

**De base de datos:** se refieren a bases de datos individuales, y por extensión, a todos los objetos que contiene cada base de datos.

**De tabla:** se aplican a tablas individuales, y por lo tanto, a todas las columnas de esas tabla.

**De columna:** se aplican a una columna en una tabla concreta.

**De rutina:** se aplican a los procedimientos almacenados. Aún no hemos visto nada sobre este tema, pero en **MySQL** se pueden almacenar procedimietos consistentes en varias consultas SQL.

### Crear usuarios

Aunque en la versión 5.0.2 de **MySQL** existe una sentencia para crear usuarios, **CREATE USER**, en versiones anteriores se usa exclusivamente la sentencia **GRANT** para crearlos.

En general es preferible usar **GRANT**, ya que si se crea un usuario mediante **CREATE USER**, posteriormente hay que usar una sentencia **GRANT** para concederle privilegios.

Usando **GRANT** podemos crear un usuario y al mismo tiempo concederle también los privilegios que tendrá. La sintaxis simplificada que usaremos para **GRANT**, sin preocuparnos de temas de cifrados seguros que dejaremos ese tema para capítulos avanzados, es:

```
GRANT priv_type [(column_list)] [, priv_type [(column_list)]] ...
  ON {tbl_name | * | *.* | db_name.*}
  TO user [IDENTIFIED BY [PASSWORD] 'password']
      [, user [IDENTIFIED BY [PASSWORD] 'password']] ...
```

La primera parte *priv\_type [(column\_list)]* permite definir el tipo de privilegio concedido para determinadas columnas. La segunda *ON {tbl\_name | \* | \*.\* | db\_name.\*}*, permite conceder privilegios en niveles globales, de base de datos o de tablas.

Para crear un usuario sin privilegios usaremos la sentencia:

```
mysql> GRANT USAGE ON *.* TO anonimo IDENTIFIED BY 'clave';
Query OK, 0 rows affected (0.02 sec)
```

Hay que tener en cuenta que la contraseña se debe introducir entre comillas de forma obligatoria.

Un usuario 'anonimo' podrá abrir una sesión **MySQL** mediante una orden:

```
C:\mysql -h localhost -u anonimo -p
```

Pero no podrá hacer mucho más, ya que no tiene privilegios. No tendrá, por ejemplo, oportunidad de hacer selecciones de datos, de crear bases de datos o tablas, insertar datos, etc.

## Conceder privilegios

Para que un usuario pueda hacer algo más que consultar algunas variables del sistema debe tener algún privilegio. Lo más simple es conceder el privilegio para seleccionar datos de una tabla concreta. Esto

se haría así:

La misma sentencia GRANT se usa para añadir privilegios a un usuario existente.

```
mysql> GRANT SELECT ON prueba.gente TO anonimo;
Query OK, 0 rows affected (0.02 sec)
```

Esta sentencia concede al usuario 'anonimo' el privilegio de ejecutar sentencias SELECT sobre la tabla 'gente' de la base de datos 'prueba'.

Un usuario que abra una sesión y se identifique como 'anonimo' podrá ejecutar estas sentencias:

```
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| prueba   |
+-----+
1 row in set (0.01 sec)

mysql> USE prueba;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_prueba |
+-----+
| gente             |
+-----+
1 row in set (0.00 sec)

mysql> SELECT * FROM gente;
+-----+-----+
| nombre  | fecha      |
+-----+-----+
| Fulano  | 1985-04-12 |
| Mengano | 1978-06-15 |
| Tulano  | 2001-12-02 |
| Pegano  | 1993-02-10 |
| Pimplano | 1978-06-15 |
| Frutano | 1985-04-12 |
+-----+-----+
6 rows in set (0.05 sec)
```

```
mysql>
```

Como se ve, para este usuario sólo existe la base de datos 'prueba' y dentro de esta, la tabla 'gente'. Además, podrá hacer consultas sobre esa tabla, pero no podrá añadir ni modificar datos, ni por supuesto, crear o destruir tablas ni bases de datos.

Para conceder privilegios globales se usa *ON \*.\**, para indicar que los privilegios se conceden en todas las tablas de todas las bases de datos.

Para conceder privilegios en bases de datos se usa *ON nombre\_db.\**, indicando que los privilegios se conceden sobre todas las tablas de la base de datos 'nombre\_db'.

Usando *ON nombre\_db.nombre\_tabla*, concedemos privilegios de nivel de tabla para la tabla y base de datos especificada.

En cuanto a los privilegios de columna, para concederlos se usa la sintaxis *tipo\_privilegio (lista\_de\_columnas), [tipo\_privilegio (lista\_de\_columnas)]*.

Otros privilegios que se pueden conceder son:

- **ALL:** para conceder todos los privilegios.
- **CREATE:** permite crear nuevas tablas.
- **DELETE:** permite usar la sentencia [DELETE](#).
- **DROP:** permite borrar tablas.
- **INSERT:** permite insertar datos en tablas.
- **UPDATE:** permite usar la sentencia [UPDATE](#).

Para ver una lista de todos los privilegios existentes consultar la sintaxis de la sentencia [GRANT](#).

Se pueden conceder varios privilegios en una única sentencia. Por ejemplo:

```
mysql> GRANT SELECT, UPDATE ON prueba.gente TO anonimo IDENTIFIED BY
'clave';
Query OK, 0 rows affected (0.22 sec)

mysql>
```

Un detalle importante es que para crear usuarios se debe tener el privilegio *GRANT OPTION*, y que sólo se pueden conceder privilegios que se posean.

## Revocar privilegios

Para revocar privilegios se usa la sentencia [REVOKE](#).

```
REVOKE priv_type [(column_list)] [, priv_type [(column_list)]] ...
  ON {tbl_name | * | *.* | db_name.*}
  FROM user [, user] ...
```

La sintaxis es similar a la de [GRANT](#), por ejemplo, para revocar el privilegio *SELECT* de nuestro usuario 'anonimo', usaremos la sentencia:

```
mysql> REVOKE SELECT ON prueba.gente FROM anonimo;
Query OK, 0 rows affected (0.05 sec)
```

## Mostrar los privilegios de un usuario

Podemos ver qué privilegios se han concedido a un usuario mediante la sentencia [SHOW GRANTS](#). La salida de esta sentencia es una lista de sentencias [GRANT](#) que se deben ejecutar para conceder los privilegios que tiene el usuario. Por ejemplo:

```
mysql> SHOW GRANTS FOR anonimo;
+-----+-----+-----+-----+-----+-----+
| Grants for anonimo@%                                     |
+-----+-----+-----+-----+-----+-----+
| GRANT USAGE ON *.* TO 'anonimo'@'%' IDENTIFIED BY PASSWORD '*5...' |
| GRANT SELECT ON `prueba`.`gente` TO 'anonimo'@'%'      |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

## Nombres de usuarios y contraseñas

Como podemos ver por la salida de la sentencia [SHOW GRANTS](#), el nombre de usuario no se limita a un nombre simple, sino que tiene dos partes. La primera consiste en un nombre de usuario, en nuestro ejemplo 'anonimo'. La segunda parte, que aparece separada de la primera por el carácter '@' es un nombre de máquina (host). Este nombre puede ser bien el de una máquina, por ejemplo, 'localhost'

para referirse al ordenador local, o cualquier otro nombre, o bien una ip.

La parte de la máquina es opcional, y si como en nuestro caso, no se pone, el usuario podrá conectarse desde cualquier máquina. La salida de [SHOW GRANTS](#) lo indica usando el comodín '%' para el nombre de la máquina.

Si creamos un usuario para una máquina o conjunto de máquinas determinado, ese usuario no podrá conectar desde otras máquinas. Por ejemplo:

```
mysql> GRANT USAGE ON * TO anonimo@localhost IDENTIFIED BY 'clave';
Query OK, 0 rows affected (0.00 sec)
```

Un usuario que se identifique como 'anonimo' sólo podrá entrar desde el mismo ordenador donde se está ejecutando el servidor.

En este otro ejemplo:

```
mysql> GRANT USAGE ON * TO anonimo@10.28.56.15 IDENTIFIED BY 'clave';
Query OK, 0 rows affected (0.00 sec)
```

El usuario 'anonimo' sólo puede conectarse desde un ordenador cuyo IP sea '10.28.56.15'.

Aunque asignar una contraseña es opcional, por motivos de seguridad es recomendable asignar siempre una.

La contraseña se puede escribir entre comillas simples cuando se crea un usuario, o se puede usar la salida de la función [PASSWORD\(\)](#) de forma literal, para evitar enviar la clave en texto legible.

Si al añadir privilegios se usar una clave diferente en la cláusula *IDENTIFIED BY*, sencillamente se sustituye la contraseña por la nueva.

## Borrar usuarios

Para eliminar usuarios se usa la sentencia [DROP USER](#).

No se puede eliminar un usuario que tenga privilegios, por ejemplo:

```
mysql> DROP USER anonimo;
ERROR 1268 (HY000): Can't drop one or more of the requested users
mysql>
```

Para eliminar el usuario primero hay que revocar todos sus privilegios:

```
mysql> SHOW GRANTS FOR anonimo;
+-----+
| Grants for anonimo@% |
+-----+
| GRANT USAGE ON *.* TO 'anonimo'@'%' IDENTIFIED BY PASSWORD '*5...' |
| GRANT SELECT ON `prueba`.`gente` TO 'anonimo'@'%' |
+-----+
2 rows in set (0.00 sec)

mysql> REVOKE SELECT ON prueba.gente FROM anonimo;
Query OK, 0 rows affected (0.00 sec)

mysql> DROP USER anonimo;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

# 14 Lenguaje SQL

## Importar y exportar datos

MySQL permite copiar tablas en diferentes formatos de texto, así como importar datos a partir de fichero de texto en diferentes formatos.

Esto se puede usar para exportar los datos de nuestras bases de datos a otras aplicaciones, o bien para importar datos desde otras fuentes a nuestras tablas. También se puede usar para hacer copias de seguridad y restaurarlas posteriormente.

### Exportar a otros ficheros

Para extraer datos desde una base de datos a un fichero se usa la sentencia SELECT ... INTO OUTFILE.

El resto de las cláusulas de SELECT siguen siendo aplicables, la única diferencia es que la salida de la selección se envía a un fichero en lugar de hacerlo a la consola.

La sintaxis de la parte *INTO OUTFILE* es:

```
[INTO OUTFILE 'file_name' export_options]
```

*file\_name* es el nombre del fichero de salida. Ese fichero no debe existir, ya que en caso contrario la sentencia fallará.

En cuanto a las opciones de exportación son las mismas que para las cláusulas *FIELDS* y *LINES* de LOAD DATA. Su sintaxis es:

```
[FIELDS
  [TERMINATED BY '\t']
  [[OPTIONALLY] ENCLOSED BY '']
  [ESCAPED BY '\\']
]
[LINES
  [STARTING BY '']
  [TERMINATED BY '\n']
]
```

Estas cláusulas nos permiten crear diferentes formatos de ficheros de salida.

La cláusula *FIELDS* se refiere a las opciones de cada columna:

- *TERMINATED BY 'carácter'*: nos permite elegir el carácter delimitador que se usará para selarar cada columna. Por defecto, el valor que se usa es el tabulador, pero podemos usar ';', ',', etc.
- *[OPTIONALLY] ENCLOSED BY 'carácter'*: sirve para elegir el carácter usado para entrecomillar cada columna. Por defecto no se entrecomilla ninguna columna, pero podemos elegir cualquier carácter. Si se añade la palabra *OPTIONALLY* sólo se entrecomillarán las columnas de texto y fecha.
- *ESCAPED BY 'carácter'*: sirve para indicar el carácter que se usará para escapar aquellos caracteres que pueden dificultar la lectura posterior del fichero. Por ejemplo, si terminamos las columnas con ',' y no las entrecomillamos, un carácter ',' dentro de una columna de texto se interpretará como un separador de columnas. Para evitar esto se puede escapar esa coma con otro carácter. Por defecto se usa el carácter '\'

La cláusula *LINES* se refiere a las opciones para cada fila:

- *STARTING BY 'carácter'*: permite seleccionar el carácter para comenzar cada línea. Por defecto no se usa ningún carácter para ello.
- *TERMINATED BY 'carácter'*: permite elegir el carácter para terminar cada línea. Por defecto es el retorno de línea, pero se puede usar cualquier otro carácter o caracteres, por ejemplo '\r\n'.

Por ejemplo, para obtener un fichero de texto a partir de la tabla 'gente', con las columnas delimitadas por ';', entrecomillando las columnas de texto con '"' y separando cada fila por la secuencia '\r\n', usaremos la siguiente sentencia:

```
mysql> SELECT * FROM gente
-> INTO OUTFILE "gente.txt"
-> FIELDS TERMINATED BY ';'
-> OPTIONALLY ENCLOSED BY '\"'
-> LINES TERMINATED BY '\n\r';
Query OK, 5 rows affected (0.00 sec)

mysql>
```

El fichero de salida tendrá este aspecto:

```
"Fulano" ; "1974-04-12"
```

```
"Mengano" ; "1978-06-15"
"Tulano" ; "2000-12-02"
"Pegano" ; "1993-02-10"
"Mengano" ; \N
```

La fecha para "Mengano" era *NULL*, para indicarlo se muestra el valor \N.

## Importar a partir de ficheros externos

Por supuesto, el proceso contrario también es posible. Podemos leer el contenido de un fichero de texto en una tabla. El fichero origen puede haber sido creado mediante una sentencia [SELECT ... INTO OUTFILE](#), o mediante cualquier otro medio.

Para hacerlo disponemos de la sentencia [LOAD DATA](#), cuya sintaxis más simple es:

```
LOAD DATA [LOCAL] INFILE 'file_name.txt'
  [REPLACE | IGNORE]
  INTO TABLE tbl_name
  [FIELDS
    [TERMINATED BY '\t']
    [[OPTIONALLY] ENCLOSED BY '']
    [ESCAPED BY '\\']
  ]
  [LINES
    [STARTING BY '']
    [TERMINATED BY '\n']
  ]
  [IGNORE number LINES]
  [(col_name, ...)]
```

La cláusula *LOCAL* indica, si aparece, que el fichero está en el ordenador del cliente. Si no se especifica el fichero de texto se buscará en el servidor, concretamente en el mismo directorio donde esté la base de datos. Esto nos permite importar datos desde nuestro ordenador en un sistema en que el servidor de **MySQL** se encuentra en otra máquina.

Las cláusulas *REPLACE* e *IGNORE* afectan al modo en que se tratan las filas leídas que contengan el mismo valor para una clave principal o única para una fila existente en la tabla. Si se especifica *REPLACE* se sustituirá la fila actual por la leída. Si se especifica *IGNORE* el valor leído será ignorado.

La parte *INTO TABLA tbl\_name* indica en qué tabla se insertarán los valores leídos.

No comentaremos mucho sobre las cláusulas *FIELDS* y *LINES* ya que su significado es el mismo que vimos para la sentencia [SELECT ... INTO OUTFILE](#). Estas sentencias nos permiten interpretar correctamente cada fila y cada columna, adaptándonos al formato del fichero de texto de entrada.

La misma utilidad tiene la cláusula *IGNORE número LINES*, que nos permite que las primeras *número* líneas no se interpreten como datos a importar. Es frecuente que los ficheros de texto que usaremos como fuente de datos contengan algunas cabeceras que expliquen el contenido del fichero, o que contengan los nombres de cada columna. Usando esta cláusula podemos ignorarlas.

La última parte nos permite indicar la columna a la que será asignada cada una de las columnas leídas, esto será útil si el orden de las columnas en la tabla no es el mismo que en el fichero de texto, o si el número de columnas es diferente en ambos.

Por ejemplo, supongamos que queremos añadir el contenido de este fichero a la tabla "gente":

```
Fichero de datos de "gente"
fecha,nombre
2004-03-15,Xulana
2000-09-09,Con Clase
1998-04-15,Pingrana
```

Como vemos, hay dos filas al principio que no contienen datos válidos, las columnas están separadas con comas y, como hemos editado el fichero con el "notepad", las líneas terminan con "\n\r". La sentencia adecuada para leer los datos es:

```
mysql> LOAD DATA INFILE "gente.txt"
-> INTO TABLE gente
-> FIELDS TERMINATED BY ','
-> LINES TERMINATED BY '\r\n'
-> IGNORE 2 LINES
-> (fecha,nombre);
Query OK, 3 rows affected (0.00 sec)
Records: 3 Deleted: 0 Skipped: 0 Warnings: 0

mysql>
```

El nuevo contenido de la tabla es:

```
mysql> SELECT * FROM gente;
+-----+-----+
```

nombre	fecha
Fulano	1974-04-12
Mengano	1978-06-15
Tulano	2000-12-02
Pegano	1993-02-10
Mengano	NULL
Xulana	2004-03-15
Con Clase	2000-09-09
Pingrana	1998-04-15

+-----+-----+

8 rows in set (0.00 sec)

mysql>

## **Apendice A: Instalación de MySQL**

Existen varias versiones para varias plataformas diferentes: Linux, Windows, Solaris.

Generalmente existen varias versiones distintas para cada plataforma. Siempre es posible conseguir una versión estable, que es la recomendada, alguna anterior, y la que actualmente esté en fase de desarrollo, que está destinada a personas que quieran colaborar en el desarrollo, buscando errores o probando las últimas versiones.

Siempre que sea posible hay que elegir la versión recomendada.

Daremos una guía para la instalación en Windows, y esperaremos colaboraciones desinteresadas para otras plataformas.

### **Instalación en Windows**

En el momento de escribir estas líneas, la versión recomendada es la 4.1.9. El fichero que hay que descargar se llama "mysql-4.1.9-win32.zip". Se trata de un fichero zip que contiene un fichero de instalación "setup.exe".

Después de descargarlo se descomprime y se ejecuta el fichero "setup.exe".

El proceso de instalación está muy mejorado con respecto a versiones anteriores, y bastará con seguir las indicaciones en cada pantalla.

### **Instalación en Linux**

En preparación.

### **Instalación en Solaris**

En preparación.

## Apendice B: Reglas para nombres de bases de datos, tablas, índices, columnas y alias

*Este apéndice es una traducción del manual de MySQL.*

Los nombres de bases de datos, tablas, índices, columnas y alias son identificadores. Esta sección describe la sintaxis permitida para crear identificadores en **MySQL**.

La tabla siguiente describe la longitud máxima y los caracteres permitidos para cada tipo de identificador.

Identificador	Longitud máxima (bytes)	Caracteres permitidos
Base de datos	64	Cualquier carácter permitido en un nombre de directorio, excepto '/', '\' o '.'
Tabla	64	Cualquier carácter permitido para un nombre de fichero, excepto '/', '\' o '.'
Columna	64	Todos los caracteres
Índice	64	Todos los caracteres
Alias	255	Todos los caracteres

Como añadido a las restricciones comentadas en la tabla, ningún identificador puede contener el valor ASCII 0 o un byte con el valor 255. Los nombres de bases de datos, tablas y columnas no pueden terminar con espacios. Antes de MySQL 4.1, los caracteres de comillas no deben ser usados en los identificadores.

A partir de MySQL 4.1, los identificadores se almacenan usando Unicode (UTF8). Esto se aplica a los identificadores de definiciones de tabla que se almacenan en ficheros '.frm' y a identificadores almacenados en las tablas de privilegios en la base de datos mysql. Aunque los identificadores Unicode pueden incluir caracteres multi-byte, hay que tener en cuenta que las longitudes máximas mostradas en la tabla se cuentan en bytes. Si un identificador contiene caracteres multi-byte, el número de caracteres permitidos para él será menor que el valor mostrado en la tabla.

Un identificador puede estar o no entrecomillado. Si un identificador es una palabra reservada o contiene caracteres especiales, se debe entrecomillar cada vez que sea referenciado. Para ver una lista de las palabras reservadas, ver la sección [palabras reservadas](#). Los caracteres especiales son aquellos que están fuera del conjunto de caracteres alfanuméricos para el conjunto de caracteres actual, '\_' y '\$'.

El carácter de entrecomillado de identificadores es la tilde izquierda (``):

```
mysql> SELECT * FROM `select` WHERE `select`.id > 100;
```

Si el modo del servidor SQL incluye la opción de modo ANSI\_QUOTES, también estará permitido entrecomillar identificadores con comillas dobles:

```
mysql> CREATE TABLE "test" (col INT);
ERROR 1064: You have an error in your SQL syntax. (...)
mysql> SET sql_mode='ANSI_QUOTES';
mysql> CREATE TABLE "test" (col INT);
Query OK, 0 rows affected (0.00 sec)
```

Desde MySQL 4.1, los caracteres de entrecomillado de identificadores pueden ser incluidos en el interior de un identificador si éste se entrecomilla. Si el carácter a incluir dentro del identificador es el mismo que se usa para entrecomillar el propio identificador, hay que duplicar el carácter. La siguiente sentencia crea una tabla con el nombre a``b`` que contiene una columna llamada c``d``:

```
mysql> CREATE TABLE `a``b` (`c``d` INT);
```

El entrecomillado de identificadores se introdujo en MySQL 3.23.6 para permitir el uso de identificadores que sean palabras reservadas o que contengan caracteres especiales. Antes de la versión 3.23.6, no es posible usar identificadores que requieran entrecomillado, de modo que las reglas para los identificadores legales son más restrictivas:

- Un nombre debe consistir en caracteres alfanuméricos del conjunto de caracteres actual, '\_' y '\$'. El conjunto de caracteres por defecto es ISO-8859-1 (Latin1). Esto puede ser cambiado con la opción --default-character-set de *mysqld*.
- Un nombre puede empezar con cualquier carácter que sea legal en el nombre. En particular, un nombre puede empezar con un dígito; esto difiere de muchos otros sistemas de bases de datos. Sin embargo, un nombre sin entrecomillar no puede consistir sólo de dígitos.
- No se puede usar el carácter '.' en nombres ya que se usa para para extender el formato con el que se puede hacer referencia a columnas.

Es recomendable no usar nombres como 1e, porque una expresión como 1e+1 es ambigua. Puede ser interpretada como la expresión 1e + 1 o como el número 1e+1, dependiendo del contexto.

## Calificadores de identificadores

**MySQL** permite nombres que consisten en un único identificador o en múltiples identificadores. Los componentes de un nombre compuesto deben estar separados por un punto ('.'). Las partes iniciales de un nombre compuesto actúan como calificadores que afectan al contexto dentro de cual, se interpreta el identificador final.

En **MySQL** se puede hacer referencia a una columna usando cualquiera de las formas siguientes:

Referencia de columna	Significado
nombre_columna	La columna nombre_columna de cualquiera de las tablas usadas en la consulta que contenga una columna con ese nombre.
nombre_tabla.nombre_columna	La columna nombre_columna de la tabla nombre_tabla de la base de datos por defecto.
nombre_basedatos.nombre_tabla.nombre_columna	La columna nombre_columna de la tabla nombre_tabla de la base de datos nombre_basedatos. Esta sintaxis no está disponible antes de MySQL 3.22.

Si cualquiera de los componentes de un nombre con varias partes requiere entrecomillado, hay que entrecomillar cada uno individualmente en lugar de entrecomillarlo completo. Por ejemplo, `mi-tabla`.`mi-columna` es legal, sin embargo `mi-tabla.mi-columna` no.

No será necesario especificar un prefijo de nombre de tabla o de base de datos para una referencia de columna en una sentencia a no ser que la referencia pueda ser ambigua. Supongamos que las tablas t1 y t2 contienen cada una una columna c, y se quiere recuperar c en una sentencia **SELECT** que usa ambas tablas. En ese caso, c es ambiguo porque no es único entre las dos tablas usadas. Se debe calificar con el nombre de la tabla como t1.c o t2.c para indicar a que tabla nos referimos. De modo similar, para recuperar datos desde una tabla t en una base de datos db1 y desde una tabla t en una base de datos db2 en la misma sentencia, se debe hacer referencia a las columnas en esas tablas como db1.t.nombre\_columna y db2.t.nombre\_columna.

La sintaxis .nombre\_tabla significa la tabla nombre\_tabla en la base de datos actual. Esta sintaxis se acepta por compatibilidad con ODBC ya que algunos programas ODBC usan el '.' como prefijos para nombres de tablas.

## Sensibilidad al tipo

En **MySQL**, las bases de datos corresponden a directorios dentro del directorio de datos "data". A las tablas dentro de una base de datos les corresponde, por lo menos, un fichero dentro del directorio de la

base de datos (y posiblemente más, dependiendo del motor de almacenamiento). En consecuencia, la distinción entre mayúsculas y minúsculas que haga el sistema operativo determinará la distinción que se haga en los nombres de bases de datos y de tablas. Esto significa que los nombres de bases de datos y tablas no son sensibles al tipo en **Windows**, y sí lo son en la mayor parte de las variantes de **Unix**. Una excepción notable es **Mac OS X**, que está basado en **Unix** pero usa un sistema de ficheros por defecto del tipo (HFS+) que no es sensible al tipo. Sin embargo, **Mac OS X** también soporta volúmenes UFS, los cuales son sensibles al tipo lo mismo que cualquier **Unix**.

**Nota:** aunque los nombres de bases de datos y tablas no sean sensibles al tipo en algunas plataformas, no se debe hacer referencia a una base de datos o tabla dada usando diferentes tipos en la misma consulta. La siguiente consulta no funciona porque se refiere a una tabla dos veces, una como `my_table` y otra como `MY_TABLE`:

```
mysql> SELECT * FROM my_table WHERE MY_TABLE.col=1;
```

Los nombres de columna, índices y alias de columna no son sensibles al tipo en ninguna plataforma.

Los alias de tabla son sensibles al tipo antes de MySQL 4.1.1. La siguiente consulta no funcionará porque se refiere a un alias como `a` y como `A`:

```
mysql> SELECT col_name FROM tbl_name AS a
-> WHERE a.col_name = 1 OR A.col_name = 2;
```

Si hay problemas para recordar el tipo de letras permitido para nombres de bases de datos y tablas, es mejor adoptar una convención rígida, como crear siempre las bases de datos y tablas con nombres que sólo contengan caracteres en minúsculas.

El modo en que se almacenan los nombres de tablas y bases de datos en disco y son usados en **MySQL** depende de la definición de la variable de sistema `lower_case_table_names`, la cual se puede asignar cuando se arranca `mysqld`. `lower_case_table_names` puede tomar uno de los valores siguientes:

Valor	Significado
0	Los nombres de tablas y bases de datos se almacenan en disco usando los tipos de caracteres usados en en las sentencias <code>CREATE TABLE</code> o <code>CREATE DATABASE</code> . Las comparaciones de nombres son sensibles al tipo. Este es el valor en sistemas <b>Unix</b> . Si se fuerza este valor 0 con <code>--lower-case-table-names=0</code> en un sistema de ficheros no sensible al tipo y se accede a nombres de tablas <b>MyISAM</b> usando diferentes tipos de caracteres, se puede producir corrupción de índices.

1	Los nombres de las tablas se almacenan en disco usando minúsculas y las comparaciones de nombres no son sensibles al tipo. <b>MySQL</b> convierte todos los nombres de tablas a minúsculas al almacenarlos y leerlos. Este comportamiento también se aplica a nombres de bases de datos desde MySQL 4.0.2, y a alias de tablas a partir de 4.1.1. Este es el valor por defecto en sistemas <b>Windows</b> y <b>Mac OS X</b> .
2	Los nombres de tablas y bases de datos se almacenan en disco usando el tipo de letra especificado en las sentencias <b>CREATE TABLE</b> o <b>CREATE DATABASE</b> , pero <b>MySQL</b> los convierte a minúscula al leerlos. Las comparaciones de nombres no son sensibles al tipo. Nota: esto funciona sólo en sistemas de ficheros que no sean sensibles al tipo. Los nombres de tablas <b>InnoDB</b> se almacenan en minúsculas, igual que si <code>lower_case_table_names=1</code> . Asignar <code>lower_case_table_names</code> a 2 se puede hacer desde MySQL 4.0.18.

Si sólo se está usando **MySQL** para una plataforma, generalmente no será necesario modificar la variable `lower_case_table_names`. Sin embargo, se pueden presentar dificultades si se quieren transferir tablas entre plataformas que tengan sistemas de ficheros con distintas sensibilidades al tipo. Por ejemplo, en **Unix**, se pueden tener dos tablas diferentes llamadas `my_table` y `MY_TABLE`, pero en **Windows** estos nombres se consideran el mismo. Para impedir problemas de transferencia de datos debidos al tipo de letras usados en nombres de bases de datos o tablas, hay dos opciones:

- Usar `lower_case_table_names=1` en todos los sistemas. La desventaja principal con esta opción es que cuando se usa **SHOW TABLES** o **SHOW DATABASES**, no se ven los nombres en el tipo de letras original.
- Usar `lower_case_table_names=0` en **Unix** y `lower_case_table_names=2` en **Windows**. Esto preserva el tipo de letras en nombres de bases de datos y tablas. La desventaja es que se debe asegurar que las consultas siempre se refieren a los nombres de bases de datos y tablas usando los tipos de caracteres correctos en **Windows**. Si se transfieren las consultas a **Unix**, donde el tipo de los caracteres es importante, no funcionarán si el tipo de letra es incorrecto.

**Nota:** antes de asignar 1 a `lower_case_table_names` en **Unix**, se deben convertir los viejos nombres de bases de datos y tablas a minúscula antes de reiniciar *mysqld*.

## Apéndice C: Expresiones regulares

*Este apéndice es una traducción del manual de MySQL.*

Una expresión regular es una forma muy potente de especificar un patrón para una búsqueda compleja.

**MySQL** usa la implementación de Henry Spencer para expresiones regulares, que ha sido ajustado para ceñirse a *POSIX 1003.2*. **MySQL** usa una versión extendida para soportar operaciones de coincidencia de patrones realizadas con el operador **REGEXP** en sentencias SQL.

Este apéndice es un resumen, con ejemplos, de las características especiales y de las construcciones que pueden ser usadas en **MySQL** para operaciones **REGEXP**. No contiene todos los detalles que pueden ser encontrados en el manual de regex(7) de Henry Spencer. Dicho manual está incluido en las distribuciones fuente de **MySQL**, en el fichero 'regex.7' bajo el directorio 'regex'.

Una expresión regular describe un conjunto de cadenas. La expresión regular más sencilla es aquella que no contiene caracteres especiales. Por ejemplo, la expresión regular "hola" coincide con "hola" y con nada más.

Las expresiones regulares no triviales usan ciertas construcciones especiales de modo que pueden coincidir con más de una cadena. Por ejemplo, la expresión regular "Hola|mundo" coincide tanto con la cadena "Hola" como con la cadena "mundo".

Como ejemplo algo más complejo, la expresión regular "B[an]\*s" coincide con cualquiera de las cadenas siguientes "Bananas", "Baaaaas", "Bs", y cualquier otra cadena que empiece con 'B', termine con 's', y contenga cualquier número de caracteres 'a' o 'n' entre la 'B' y la 's'.

Una expresión regular para el operador **REGEXP** puede usar cualquiera de los siguientes caracteres especiales u construcciones:

^

Coincidencia del principio de una cadena.

```
mysql> SELECT 'fo\nfo' REGEXP '^fo$';           -> 0
mysql> SELECT 'fofo' REGEXP '^fo';             -> 1
```

\$

## Coincidencia del final de una cadena.

```
mysql> SELECT 'fo\no' REGEXP '^fo\no$';      -> 1
mysql> SELECT 'fo\no' REGEXP '^fo$';        -> 0
```

## Coincidencia de cualquier carácter (incluyendo los de avance o el retorno de línea).

```
mysql> SELECT 'fofo' REGEXP '^f.*$'; -> 1
mysql> SELECT 'fo\r\nfo' REGEXP '^f.*$'; -> 1
```

a\*

## Coincidencia de cualquier secuencia de cero o más caracteres.

```
mysql> SELECT 'Ban' REGEXP '^Ba*n';          -> 1
mysql> SELECT 'Baaan' REGEXP '^Ba*n';       -> 1
mysql> SELECT 'Bn' REGEXP '^Ba*n';          -> 1
```

a+

## Coincidencia de cualquier secuencia de uno o más caracteres.

```
mysql> SELECT 'Ban' REGEXP '^Ba+n';          -> 1
mysql> SELECT 'Bn' REGEXP '^Ba+n';          -> 0
```

a?

## Coincidencia de ninguno o de un carácter.

```
mysql> SELECT 'Bn' REGEXP '^Ba?n';          -> 1
mysql> SELECT 'Ban' REGEXP '^Ba?n';         -> 1
mysql> SELECT 'Baan' REGEXP '^Ba?n';        -> 0
```

de|abc

Coincidencia de cualquiera de las secuencias "de" o "abc".

```
mysql> SELECT 'pi' REGEXP 'pi|apa';           -> 1
mysql> SELECT 'axe' REGEXP 'pi|apa';         -> 0
mysql> SELECT 'apa' REGEXP 'pi|apa';         -> 1
mysql> SELECT 'apa' REGEXP '^(pi|apa)$';     -> 1
mysql> SELECT 'pi' REGEXP '^(pi|apa)$';     -> 1
mysql> SELECT 'pix' REGEXP '^(pi|apa)$';    -> 0
```

(abc)\*

Coincidencia de ninguna o más instancias de la secuencia "abc".

```
mysql> SELECT 'pi' REGEXP '^(pi)*$';        -> 1
mysql> SELECT 'pip' REGEXP '^(pi)*$';       -> 0
mysql> SELECT 'pypi' REGEXP '^(pi)*$';     -> 1
```

{1}

{2,3}

La notación {n} o {m,n} proporciona una forma más general para escribir expresiones regulares que coincidan con muchas apariciones del átomo o trozo previo del patrón. m y n son enteros.

a\*

Puede ser escrito como "a{0,}".

a+

Puede ser escrito como "a{1,}".

a?

Puede ser escrito como "a{0,1}".

Para ser más precisos, "a{n}" coincide exactamente con n instancias de a. "a{n,}"

coincide con n o más instancias de a. "a{m,n}" coincide con un número entre m y n de instancias de a, ambos incluidos. m y n deben estar en el rango de 0 a RE\_DUP\_MAX (por defecto 255), incluidos. Si se dan tanto m como n, m debe ser menor o igual que n.

```
mysql> SELECT 'abcde' REGEXP 'a[bcd]{2}e';          -> 0
mysql> SELECT 'abcde' REGEXP 'a[bcd]{3}e';          -> 1
mysql> SELECT 'abcde' REGEXP 'a[bcd]{1,10}e';       -> 1
```

[a-dX]

[^a-dX]

Coincidencia de cualquier carácter que sea (o que no sea, si se usa ^) cualquiera entre 'a', 'b', 'c', 'd' o 'X'. Un carácter '-' entre otros dos caracteres forma un rango que incluye todos los caracteres desde el primero al segundo. Por ejemplo, [0-9] coincide con cualquier dígito decimal. Para incluir un carácter ']' literal, debe ser inmediatamente seguido por el corchete abierto '['. Para incluir el carácter '-', debe ser escrito el primero o el último. Cualquier carácter dentro de [], que no tenga definido un significado especial, coincide sólo con él mismo.

```
mysql> SELECT 'aXbc' REGEXP '[a-dXYZ]';            -> 1
mysql> SELECT 'aXbc' REGEXP '^[a-dXYZ]$';          -> 0
mysql> SELECT 'aXbc' REGEXP '^[a-dXYZ]+$';         -> 1
mysql> SELECT 'aXbc' REGEXP '^[^a-dXYZ]+$';        -> 0
mysql> SELECT 'gheis' REGEXP '^[^a-dXYZ]+$';       -> 1
mysql> SELECT 'gheisa' REGEXP '^[^a-dXYZ]+$';      -> 0
```

[.caracteres.]

En el interior de una expresión entre corchetes (escrita usando '[' y ']'), coincide con la secuencia de caracteres el elemento recopilado. "caracteres" puede ser tanto un carácter individual como un nombre de carácter, como *newline*. Se puede encontrar una lista completa de nombres de caracteres en el fichero 'regexp/cname.h'.

```
mysql> SELECT '~' REGEXP '[[.~.]]';                 -> 1
mysql> SELECT '~' REGEXP '[[.tilde.]]';             -> 1
```

[=clase\_carácter=]

En el interior de una expresión entre corchetes (escrita usando '[' y ']'),

[=clase\_carácter=] representa una equivalencia de clase. Con ella coinciden todos los caracteres con el mismo valor de colección, incluido él mismo. Por ejemplo, si 'o' y '(+)' son miembros de una clase de equivalencia, entonces "[[=o=]]", "[[=(+)=]]" y "[o(+)]" son sinónimos. Una clase de equivalencia no debe ser usada como extremo de un rango.

[:clase\_carácter:]

En el interior de una expresión entre corchetes (escrita usando '[' y ']'), [:character\_class:] representa una clase de caracteres que coincide con todos los caracteres pertenecientes a esa clase. Los nombres de clases estándar son:

alnum	Caracteres alfanuméricos
alpha	Caracteres alfabéticos
blank	Caracteres espacio
cntrl	Caracteres de control
digit	Dígitos
graph	Caracteres gráficos
lower	Caracteres alfabéticos en minúsculas
print	Caracteres gráficos o espacios
punct	Caracteres de puntuación
space	Espacio, tabulador, cambio de línea y retorno de línea
upper	Caracteres alfabéticos en mayúsculas
xdigit	Dígitos hexadecimales

Estas clases de caracteres están definidos en el manual de ctype(3). Una localización particular puede definir otros nombres de clases. Una clase de carácter no debe ser usada como extremo de un rango.

```
mysql> SELECT 'justalnums' REGEXP '[:alnum:]+';      -> 1
mysql> SELECT '!!!' REGEXP '[:alnum:]+';          -> 0
```

[[:<:]]

[[:>:]]

Estos marcadores señalan los límites de palabras. Son coincidencias con el principio o final de palabras, respectivamente. Una palabra es una secuencia de caracteres de

palabra que no esté precedida o seguida por caracteres de palabra. Un carácter de palabra es un carácter alfanumérico de la clase *alnum* o un carácter de subrayado ( ).

```
mysql> SELECT 'a word a' REGEXP '[[<:]]word[[>:]]';    -> 1
mysql> SELECT 'a xword a' REGEXP '[[<:]]word[[>:]]';    -> 0
```

Para usar una instancia literal de un carácter especial en una expresión regular, hay que precederlo por dos caracteres de barra de bajada (`\`). El analizador sintáctico de **MySQL** interpreta una de las barras, y la librería de expresiones regulares interpreta la otra. Por ejemplo, para buscar la coincidencia de la cadena "1+2" que contiene el carácter especial '+', sólo la última de las siguientes expresiones regulares es correcta:

```
mysql> SELECT '1+2' REGEXP '1+2';                        -> 0
mysql> SELECT '1+2' REGEXP '1\+2';                       -> 0
mysql> SELECT '1+2' REGEXP '1\\+2';                      -> 1
```

## Apéndice D: Husos horarios

*Este apéndice es una traducción del manual de MySQL.*

Antes de MySQL 4.1.3, se puede cambiar el huso horario para el servidor con la opción `--timezone=timezone_name` de `mysqld_safe`. También se puede cambiar mediante la variable de entorno `TZ` antes de arrancar `mysqld`.

Los valores permitidos para `--timezone` o `TZ` son dependientes del sistema. Hay que consultar la documentación del sistema operativo para ver qué valores son válidos.

A partir de MySQL 4.1.3, el servidor mantiene ciertos valores de huso horario:

- El huso horario del sistema. Cuando el servidor arranca, intenta determinar el huso horario del ordenador cliente y lo usa para asignar el valor de la variable de sistema `system_time_zone`.
- El huso horario actual del servidor. La variable global de sistema `time_zone system` indica el huso horario del servidor actual en el que se está operando. El valor inicial es 'SYSTEM', que indica que el huso horario del servidor es el mismo que el del sistema. El valor inicial puede ser especificado explícitamente con la opción `--default-time-zone=timezone`. Si se posee el privilegio `SUPER`, se puede asignar el valor global durante la ejecución con la sentencia:

```
mysql> SET GLOBAL time_zone = timezone;
```

- Husos horarios por conexión. Cada cliente que se conecta tiene su propio huso horario asignado, dado por la variable de sesión `time_zone`. Inicialmente su valor es el mismo que el de la variable global `time_zone`, pero se puede modificar con esta sentencia:

```
mysql> SET time_zone = timezone;
```

Los valores actuales de los husos horarios global y por conexión pueden ser recuperados de este modo:

```
mysql> SELECT @@global.time_zone, @@session.time_zone;
```

Los valores `timezone` pueden ser dados como cadenas que indiquen un desplazamiento a partir de UTC, como por ejemplo '+10:00' o '-6:00'. Si las tablas de tiempo relacionadas con husos horarios de la base de datos `mysql` han sido creados y asignados, se pueden usar también nombres de husos

horarios, como por ejemplo 'Europe/Helsinki', 'US/Eastern' o 'MET'. El valor 'SYSTEM' indica que el huso horario debe ser el mismo que el del sistema. Los nombres de husos horarios no son sensibles al tipo de carácter.

El proceso de instalación de **MySQL** crea las tablas de husos horarios en la base de datos mysql, pero no las carga. Esto se debe hacer de forma manual. (Si se está actualizando a MySQL 4.1.3 o superior desde una versión anterior, se deben crear las tablas mediante una actualización de la base de datos mysql).

Si el sistema tiene su propia base de datos de información horaria (el conjunto de ficheros que describen los husos horarios), se debe usar el programa `mysql_tzinfo_to_sql` para llenar las tablas de husos horarios. Ejemplos de tales sistemas son **Linux**, **FreeBSD**, **Sun Solaris** y **Mac OS X**. Una localización probable de estos ficheros es el directorio '/usr/share/zoneinfo'. Si el sistema no tiene una base de datos de husos horarios, se puede usar el paquete descargable descrito más abajo.

El programa `mysql_tzinfo_to_sql` se usa para cargar los valores de las tablas de husos horarios. En la línea de comandos, hay que pasar el nombre de la ruta del directorio con la información de husos horarios a `mysql_tzinfo_to_sql` y enviar la salida al programa mysql. Por ejemplo:

```
shell> mysql_tzinfo_to_sql /usr/share/zoneinfo | mysql -u root mysql
```

`mysql_tzinfo_to_sql` lee los ficheros de husos horarios del sistema y genera sentencias SQL a partir de ellos. mysql procesa esas sentencias para cargar las tablas de husos horarios.

`mysql_tzinfo_to_sql` también se usa para cargar un único fichero de huso horario, y para generar información adicional.

Para cargar un fichero `tz_file` que corresponda a un huso horario llamado `tz_name`, hay que invocar a `mysql_tzinfo_to_sql` de este modo:

```
shell> mysql_tzinfo_to_sql tz_file tz_name | mysql -u root mysql
```

Si el huso horario precisa tener en cuenta intervalos de segundos, hay que inicializar la información de intervalo de segundos de este modo, donde `tz_file` es el nombre del fichero:

```
shell> mysql_tzinfo_to_sql --leap tz_file | mysql -u root mysql
```

Si el sistema no tiene base de datos de husos horarios (por ejemplo, **Windows** o **HP-UX**), se puede

usar el paquete de tablas de husos horarios prediseñadas que está disponible para su descarga en <http://dev.mysql.com/downloads/timezones.html>. Este paquete contiene los ficheros '.frm', '.MYD' y '.MYI' para las tablas de husos horarios **MyISAM**. Estas tablas deben pertenecer a la base de datos mysql, de modo que deben colocarse esos ficheros en el subdirectorio 'mysql' de directorio de datos del servidor MySQL. El servidor debe ser detenido mientras se hace esto.

¡Cuidado! no usar el paquete descargable si el sistema tiene una base de datos de husos horarios. Usar la utilidad `mysql_tzinfo_to_sql` en su lugar. En caso contrario, se puede provocar una diferencia en la manipulación de tiempos entre **MySQL** y otras aplicaciones del sistema.

## Apéndice E Palabras reservadas en MySQL

*Este apéndice es una traducción del manual de MySQL.*

Un problema frecuente se deriva del intento de usar como identificador de una tabla o columna un nombre que se usa internamente por **MySQL** como nombre de dato o función, como *TIMESTAMP* o *GROUP*. Está permitido hacer esto (por ejemplo, *ABS* está permitido como nombre de columna). Sin embargo, por defecto, no se permiten espacios en blanco en las llamadas a función entre el nombre de la función y el paréntesis '('. Esta característica permite distinguir una llamada a función de una referencia a un nombre de columna.

Un efecto secundario de este comportamiento es que la omisión de un espacio en algunos contextos haga que un identificador sea interpretado como un nombre de función. Por ejemplo, esta sentencia es legal:

```
mysql> CREATE TABLE abs (val INT);
```

Pero si se omite el espacio después de 'abs' se produce un error de sintaxis porque entonces la sentencia parece que invoque a la función [ABS\(\)](#):

```
mysql> CREATE TABLE abs(val INT);
```

Si el modo del servidor SQL incluye el valor de modo `IGNORE_SPACE`, el servidor permite que las llamadas a función puedan tener espacios entre el nombre de la función y el paréntesis. Esto hace que los nombres de las funciones se traten con palabras reservadas. Como resultado, los identificadores que sean iguales que nombres de funciones deben ser entrecomillados como se describe en el apéndice [reglas para nombres](#).

Las palabras de la tabla siguiente son palabras reservadas explícitamente en **MySQL**. Muchas de ellas están prohibidas por SQL estándar como nombres de columna y/o tabla (por ejemplo, `GROUP`). Algunas son reservadas porque **MySQL** las necesita y (frecuentemente) usa un analizador sintáctico *yacc*. Una palabra reservada puede usarse como identificador si se entrecomilla.

Palabra	Palabra	Palabra
ADD	ALL	ALTER
ANALYZE	AND	AS

ASC	ASENSITIVE	BEFORE
BETWEEN	BIGINT	BINARY
BLOB	BOTH	BY
CALL	CASCADE	CASE
CHANGE	CHAR	CHARACTER
CHECK	COLLATE	COLUMN
CONDITION	CONNECTION	CONSTRAINT
CONTINUE	CONVERT	CREATE
CROSS	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	CURRENT_USER	CURSOR
DATABASE	DATABASES	DAY_HOUR
DAY_MICROSECOND	DAY_MINUTE	DAY_SECOND
DEC	DECIMAL	DECLARE
DEFAULT	DELAYED	DELETE
DESC	DESCRIBE	DETERMINISTIC
DISTINCT	DISTINCTROW	DIV
DOUBLE	DROP	DUAL
EACH	ELSE	ELSEIF
ENCLOSED	ESCAPED	EXISTS
EXIT	EXPLAIN	FALSE
FETCH	FLOAT	FOR
FORCE	FOREIGN	FROM
FULLTEXT	GOTO	GRANT
GROUP	HAVING	HIGH_PRIORITY
HOUR_MICROSECOND	HOUR_MINUTE	HOUR_SECOND
IF	IGNORE	IN
INDEX	INFILE	INNER
INOUT	INSENSITIVE	INSERT
INT	INTEGER	INTERVAL
INTO	IS	ITERATE
JOIN	KEY	KEYS
KILL	LEADING	LEAVE

LEFT	LIKE	LIMIT
LINES	LOAD	LOCALTIME
LOCALTIMESTAMP	LOCK	LONG
LOB	LONGTEXT	LOOP
LOW_PRIORITY	MATCH	MEDIUMBLOB
MEDIUMINT	MEDIUMTEXT	MIDDLEINT
MINUTE_MICROSECOND	MINUTE_SECOND	MOD
MODIFIES	NATURAL	NOT
NO_WRITE_TO_BINLOG	NULL	NUMERIC
ON	OPTIMIZE	OPTION
OPTIONALLY	OR	ORDER
OUT	OUTER	OUTFILE
PRECISION	PRIMARY	PROCEDURE
PURGE	READ	READS
REAL	REFERENCES	REGEXP
RENAME	REPEAT	REPLACE
REQUIRE	RESTRICT	RETURN
REVOKE	RIGHT	RLIKE
SCHEMA	SCHEMAS	SECOND_MICROSECOND
SELECT	SENSITIVE	SEPARATOR
SET	SHOW	SMALLINT
SONAME	SPATIA	SPECIFIC
SQL	SQLException	SQLSTATE
SQLWARNING	SQL_BIG_RESULT	SQL_CALC_FOUND_ROWS
SQL_SMALL_RESULT	SSL	STARTING
STRAIGHT_JOIN	TABLE	TERMINATED
THEN	TINYBLOB	TINYINT
TINYTEXT	TO	TRAILING
TRIGGER	TRUE	UNDO
UNION	UNIQUE	UNLOCK
UNSIGNED	UPDATE	USAGE
USE	USING	UTC_DATE

UTC_TIME	UTC_TIMESTAMP	VALUES
VARBINARY	VARCHAR	VARCHARACTER
VARYING	WHEN	WHERE
WHILE	WITH	WRITE
XOR	YEAR_MONTH	ZEROFILL

**MySQL** permite que algunas palabras reservadas sean usadas como identificadores sin entrecomillar porque mucha gente ya las usa. Ejemplos de esas palabras son las siguientes:

ACTION
BIT
DATE
ENUM
NO
TEXT
TIME
TIMESTAMP

## Apéndice F: Bibliografía

Entre la "bibliografía" consultada, citaré algunas páginas sobre el tema de bases de datos que he usado como referencia:

En la página de la [James Cook University](#) hay varios tutoriales sobre teoría de bases de datos bastante completos:

- [Modelo E-R](#)
- [Modelo relacional](#)
- [Álgebra relacional](#)
- [Normalización](#)

Universidad Nacional de Colombia

- [Bases de datos](#)

Instituto Tecnológico de La Paz

- [Bases de datos](#)

Universidad de Concepción

- [Manual sobre bases de datos](#)

[AulaClic:](#)

- [Curso de SQL](#)

# ***Tabla de contenido.***

## 0 Prólogo

- 0.1 Introducción
- 0.2 Instalar el servidor MySQL
- 0.3 Y... ¿por qué MySQL?

## 1 Definiciones

- 1.1 Dato
- 1.2 Base de datos
- 1.3 SGBD (DBMS)
- 1.4 Consulta
- 1.5 Redundancia de datos
- 1.6 Inconsistencia de datos
- 1.7 Integridad de datos

## 2 Diseño I, Modelo entidad-relación E-R

- 2.1 Modelado de bases de datos
- 2.2 Modelo Entidad-Relación
- 2.3 Definiciones
  - 2.3.1 Entidad
  - 2.3.2 Conjunto de entidades
  - 2.3.3 Atributo
  - 2.3.4 Dominio
  - 2.3.5 Relación
  - 2.3.6 Grado
  - 2.3.7 Clave
  - 2.3.8 Claves candidatas
  - 2.3.9 Clave principal
  - 2.3.10 Claves de interrelaciones
  - 2.3.11 Entidades fuertes y débiles
  - 2.3.12 Dependencia de existencia

### 2.4 Generalización

### 2.5 Especialización

## 2.6 Representación de entidades y relaciones: Diagramas

- 2.6.1 Entidad
- 2.6.2 Atributo
- 2.6.3 Interrelación
- 2.6.4 Dominio
- 2.6.5 Diagrama

## 2.7 Construir un modelo E-R

### 2.8 Proceso

### 2.9 Extensiones

### 2.10 Ejemplo 1

- 2.10.1 Identificar conjuntos de entidades
- 2.10.2 Identificar conjuntos de interrelaciones
- 2.10.3 Trazar primer diagrama
- 2.10.4 Identificar atributos
- 2.10.5 Seleccionar claves principales
- 2.10.6 Verificar el modelo

### 2.11 Ejemplo 2

- 2.11.1 Identificar conjuntos de entidades
- 2.11.2 Identificar conjuntos de interrelaciones
- 2.11.3 Trazar primer diagrama
- 2.11.4 Identificar atributos
- 2.11.5 Seleccionar claves principales
- 2.11.6 Verificar el modelo

## 3 Diseño II, Modelo relacional

### 3.1 Modelo relacional

### 3.2 Definiciones

- 3.2.1 Relación
- 3.2.2 Tupla
- 3.2.3 Atributo
- 3.2.4 Nulo (NULL)
- 3.2.5 Dominio
- 3.2.6 Modelo relacional
- 3.2.7 Cardinalidad
- 3.2.8 Grado
- 3.2.9 Esquema

3.2.10 Instancia

3.2.11 Clave

3.2.12 Interrelación

3.3 Paso del modelo E-R al modelo relacional

3.4 Manipulación de datos, álgebra relacional

3.4.1 Selección

3.4.2 Proyección

3.4.3 Producto cartesiano

3.4.4 Composición (Join)

3.4.5 Composición natural

3.4.6 Unión

3.4.7 Intersección

3.4.8 Diferencia

3.4.9 División

3.5 Integridad de datos

3.5.1 Restricciones sobre claves primarias

3.5.2 Integridad referencial

3.6 Propagación de claves

3.7 Ejemplo 1

3.8 Ejemplo 2

4 Diseño III, Normalización

4.1 Normalización

4.2 Primera forma normal (1FN)

4.3 Dependencias funcionales

4.3.1 Dependencia funcional completa

4.3.2 Dependencia funcional elemental

4.3.3 Dependencia funcional trivial

4.4 Segunda forma normal (2FN)

4.5 Dependencia funcional transitiva

4.6 Tercera forma normal (3FN)

4.7 Forma normal Boycce Codd (FNBC)

4.8 Atributos multivaluados

## 4.9 Dependencias multivaluadas

## 4.10 Cuarta forma normal (4FN)

## 4.11 Quinta forma normal (5FN)

## 4.12 Ejemplo 1

4.12.1 Primera forma normal

4.12.2 Segunda forma normal

4.12.3 Tercera forma normal

4.12.4 Forma normal de Boyce/Codd

4.12.5 Cuarta forma normal

## 4.13 Ejemplo 2

4.13.1 Primera forma normal

4.13.2 Segunda forma normal

4.13.3 Tercera forma normal

4.13.4 Forma normal de Boyce/Codd

4.13.5 Cuarta forma normal

## 4.14 Ejemplo 3

# 5 Tipos de columnas

## 5.1 Tipos de datos de cadenas de caracteres

5.1.1 CHAR

5.1.2 VARCHAR()

5.1.3 VARCHAR()

## 5.2 Tipos de datos enteros

5.2.1 TINYINT

5.2.2 BIT, BOOL, BOOLEAN

5.2.3 SMALLINT

5.2.4 MEDIUMINT

5.2.5 INT

5.2.6 INTEGER

5.2.7 BIGINT

## 5.3 Tipos de datos en coma flotante

5.3.1 FLOAT

- 5.3.2 FLOAT()
- 5.3.3 DOUBLE
- 5.3.4 DOUBLE PRECISION, REAL
- 5.3.5 DECIMAL
- 5.3.6 DEC, NUMERIC, FIXED

## 5.4 Tipos de datos para tiempos

- 5.4.1 DATE
- 5.4.2 DATETIME
- 5.4.3 TIMESTAMP
- 5.4.4 TIME
- 5.4.5 YEAR

## 5.5 Tipos de datos para datos sin tipo o grandes bloques de datos

- 5.5.1 TINYBLOB, TINYTEXT
- 5.5.2 BLOB, TEXT
- 5.5.3 MEDIUMBLOB, MEDIUMTEXT
- 5.5.4 LONGBLOB, LONGTEXT

## 5.6 Tipos enumerados y conjuntos

- 5.6.1 ENUM
- 5.6.2 SET

## 5.7 Ejemplo 1

- 5.7.1 Relación Estación
- 5.7.2 Relación Muestra

# 6 El cliente MySQL

- 6.1 Algunas consultas
- 6.2 Usuarios y privilegios

# 7 Creación de bases de datos

- 7.1 Crear una base de datos
- 7.2 Crear una tabla

- 7.2.1 Valores nulos
- 7.2.2 Valores por defecto
- 7.2.3 Claves primarias
- 7.2.4 Columnas autoincrementadas
- 7.2.5 Comentarios

## 7.3 Definición de creación

- 7.3.1 Índices
- 7.3.2 Claves foráneas

## 7.4 Opciones de tabla

- 7.4.1 Motor de almacenamiento

## 7.5 Verificaciones

## 7.6 Eliminar una tabla

## 7.7 Eliminar una base de datos

## 7.8 Ejemplo 1

## 7.9 Ejemplo 2

# 8 Inserción de datos

## 8.1 Insertción de nuevas filas

## 8.2 Reemplazar filas

## 8.3 Actualizar filas

## 8.4 Eliminar filas

## 8.5 Vaciar una tabla

# 9 Consultas

## 9.1 Forma incondicional

## 9.2 Limitar columnas: proyección

- 9.2.1 Alias

## 9.3 Mostramos filas repetidas

## 9.4 Limitar las filas: Selección

## 9.5 Agrupar filas

## 9.6 Cláusula HAVING

## 9.7 Ordenar resultados

## 9.8 Limitar el número de filas de salida

# 10 Operadores

## 10.1 Operador de asignación

## 10.2 Operadores lógicos

### 10.2.1 Operador Y

### 10.2.2 Operador O

### 10.2.3 Operador O exclusivo

### 10.2.4 Operador de negación

## 10.3 Reglas para las comparaciones de valores

## 10.4 Operadores de comparación

### 10.4.1 Operador de igualdad

### 10.4.2 Operador de igualdad con *NULL* seguro

### 10.4.3 Operador de desigualdad

### 10.4.4 Operadores de comparación de magnitud

### 10.4.5 Verificación de *NULL*

### 10.4.6 Verificar pertenencia a un rango

### 10.4.7 Elección de no nulos

### 10.4.8 Valores máximo y mínimo de una lista

### 10.4.9 Verificar conjuntos

### 10.4.10 Verificar nulos

### 10.4.11 Encontrar intervalo

## 10.5 Operadores aritméticos

### 10.5.1 Operador de adición o suma

### 10.5.2 Operador de sustracción o resta

### 10.5.3 Operador unitario menos

### 10.5.4 Operador de producto o multiplicación

### 10.5.5 Operador de cociente o división

### 10.5.6 Operador de división entera

## 10.6 Operadores de bits

### 10.6.1 Operador de bits O

### 10.6.2 Operador de bits Y

### 10.6.3 Operador de bits O exclusivo

### 10.6.4 Operador de bits de complemento

10.6.5 Operador de desplazamiento a la izquierda

10.6.6 Operador de desplazamiento a la derecha

10.6.7 Contar bits

## 10.7 Operadores de control de flujo

10.7.1 Operador *CASE*

## 10.8 Operadores para cadenas

10.8.1 Operador *LIKE*

10.8.2 Operador *NOT LIKE*

10.8.3 Operadores *REGEXP* y *RLIKE*

10.8.4 Operadores *NOT REGEXP* y *NOT RLIKE*

## 10.9 Operadores de casting

10.9.1 Operador *BINARY*

10.10 Tabla de precedencia de operadores

10.11 Paréntesis

# 11 Funciones

11.1 Funciones de control de flujo

11.2 Funciones matemáticas

11.3 Funciones de cadenas

11.4 Funciones de comparación de cadenas

11.5 Funciones de fecha

11.6 De búsqueda de texto

11.7 Funciones de casting (conversión de tipos)

11.8 Funciones de encriptado

11.9 Funciones de información

11.10 Miscelanea

11.11 De grupos

# 12 Consultas multitabla

12.1 Producto cartesiano

12.2 Composición (Join)

## 12.3 Composiciones internas

### 12.3.1 Composición interna natural

## 12.4 Composiciones externas

### 12.4.1 Composición externa izquierda

### 12.4.2 Composición externa derecha

### 12.4.3 Composiciones naturales externas

## 12.5 Unión

# 13 Usuarios y privilegios

## 13.1 Niveles de privilegios

## 13.2 Crear usuarios

## 13.3 Conceder privilegios

## 13.4 Revocar privilegios

## 13.5 Mostrar los privilegios de un usuario

## 13.6 Nombres de usuarios y contraseñas

## 13.7 Borrar usuarios

# 14 Importar y exportar datos

## 14.1 Exportar a otros ficheros

## 14.2 Importar a partir de ficheros externos

# A Instalación de MySQL

## A.1 Instalación en Windows

## A.2 Instalación en Linux

## A.3 Instalación en Solaris

# B Reglas para nombres

## B.1 Calificadores de identificadores

## B.2 Sensibilidad al tipo

C Expresiones regulares

D Husos horarios

E Palabras reservadas

F Bibliografía

# Indice de Sentencias SQL (67)

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Debido a las numerosas versiones de **MySQL** y al tiempo necesario para traducir la documentación original, las referencias de las diferentes sentencias provienen, en general, de distintas versiones de la documentación. En la columna de la derecha se indica la versión de procedencia de sentencia.

## -A-

Función	Versión	Función	Versión
<a href="#">ALTER TABLE</a>	4.0	<a href="#">ANALYZE TABLE</a>	4.0

## -B-

Función	Versión	Función	Versión
<a href="#">BACKUP TABLE</a>	4.1.1	<a href="#">BEGIN</a>	4.0
<a href="#">BEGIN WORK</a>	4.0		

## -C-

Función	Versión	Función	Versión
<a href="#">CHECK TABLE</a>	4.0	<a href="#">CHECKSUM TABLE</a>	4.1.1
<a href="#">COMMIT</a>	4.0	<a href="#">CREATE DATABASE</a>	4.1.1
<a href="#">CREATE INDEX</a>	4.1.1	<a href="#">CREATE TABLE</a>	4.1.1
<a href="#">CREATE USER</a>	5.0.2		

## -D-

Función	Versión	Función	Versión
<a href="#">DELETE</a>	4.1.1	<a href="#">DESCRIBE</a>	4.0
<a href="#">DO</a>	4.1.1	<a href="#">DROP DATABASE</a>	4.1.1
<a href="#">DROP INDEX</a>	4.1.1	<a href="#">DROP TABLE</a>	4.1.1
<a href="#">DROP USER</a>	4.1.1		

**-F-** 

Función	Versión	Función	Versión
<u>FLUSH</u>	4.0		

**-G-** 

Función	Versión	Función	Versión
<u>GRANT</u>	4.1.1		

**-H-** 

Función	Versión	Función	Versión
<u>HANDLER</u>	4.0		

**-I-** 

Función	Versión	Función	Versión
<u>INSERT</u>	4.1.1	<u>INSERT ... SELECT</u>	4.1.1
<u>INSERT DELAYED</u>	4.1.1		

**-J-** 

Función	Versión	Función	Versión
<u>JOIN</u>	4.1.1		

**-K-** 

Función	Versión	Función	Versión
<u>KILL</u>	4.0		

**-L-** 

Función	Versión	Función	Versión
<u>LOAD DATA</u>	4.0	<u>LOCK TABLES</u>	4.0

**-O-** 

Función	Versión	Función	Versión
<u>OPTIMIZE TABLE</u>	4.0		

**-R-** 

Función	Versión	Función	Versión

<u>RENAME TABLE</u>	4.1.1	<u>REPAIR TABLE</u>	4.0
<u>REPLACE</u>	4.1.1	<u>RESET</u>	4.1.1
<u>REVOKE</u>	4.1.1	<u>ROLLBACK</u>	4.0

**-S-** 

<b>Función</b>	<b>Versión</b>	<b>Función</b>	<b>Versión</b>
<u>SELECT</u>	4.1.1	<u>SET</u>	4.0
<u>SET TRANSACTION</u>	4.0	<u>SHOW</u>	4.1.1
<u>SHOW CHARACTER SET</u>	4.1.1	<u>SHOW COLLATION</u>	4.1.1
<u>SHOW COLUMNS</u>	4.1.1	<u>SHOW CREATE DATABASE</u>	4.1.1
<u>SHOW CREATE TABLE</u>	4.1.1	<u>SHOW CREATE VIEW</u>	5.0.1
<u>SHOW DATABASES</u>	4.1.1	<u>SHOW ENGINES</u>	4.1.1
<u>SHOW ERRORS</u>	4.1.1	<u>SHOW GRANTS</u>	4.1.1
<u>SHOW INDEX</u>	4.1.1	<u>SHOW INNODB STATUS</u>	4.1.1
<u>SHOW KEYS</u>	4.1.1	<u>SHOW LOGS</u>	4.1.1
<u>SHOW PRIVILEGES</u>	4.1.1	<u>SHOW PROCESSLIST</u>	4.1.1
<u>SHOW STATUS</u>	4.1.1	<u>SHOW TABLE STATUS</u>	4.1.1
<u>SHOW TABLES</u>	4.1.1	<u>SHOW VARIABLES</u>	4.1.1
<u>SHOW WARNINGS</u>	4.1.1	<u>START TRANSACTION</u>	4.0

**-T-** 

<b>Función</b>	<b>Versión</b>	<b>Función</b>	<b>Versión</b>
<u>TRUNCATE</u>	4.1.1		

**-U-** 

<b>Función</b>	<b>Versión</b>	<b>Función</b>	<b>Versión</b>
<u>UNION</u>	4.1.1	<u>UNLOCK TABLES</u>	4.0
<u>UPDATE</u>	4.1.1	<u>USE</u>	4.1.1

# ALTER TABLE

```
ALTER [IGNORE] TABLE tbl_name alter_specification [,
alter_specification ...]
```

Sintaxis para *alter\_specification*:

```
ADD [COLUMN] create_definition [FIRST | AFTER column_name ]
| ADD [COLUMN] (create_definition, create_definition,...)
| ADD INDEX [index_name] (index_col_name,...)
| ADD [CONSTRAINT [symbol]] PRIMARY KEY (index_col_name,...)
| ADD [CONSTRAINT [symbol]] UNIQUE [index_name] (index_col_name,...)
| ADD FULLTEXT [index_name] (index_col_name,...)
| ADD [CONSTRAINT [symbol]] FOREIGN KEY [index_name] (index_col_name,...)
    [reference_definition]
| ALTER [COLUMN] col_name {SET DEFAULT literal | DROP DEFAULT}
| CHANGE [COLUMN] old_col_name create_definition
    [FIRST | AFTER column_name]
| MODIFY [COLUMN] create_definition [FIRST | AFTER column_name]
| DROP [COLUMN] col_name
| DROP PRIMARY KEY
| DROP INDEX index_name
| DISABLE KEYS
| ENABLE KEYS
| RENAME [TO] new_tbl_name
| ORDER BY col
| CHARACTER SET character_set_name [COLLATE collation_name]
| table_options
```

**ALTER TABLE** permite modificar la estructura de una tabla existente. Por ejemplo, se pueden añadir o eliminar columnas, crear y destruir índices, cambiar el tipo de una columna existente o renombrar columnas o la propia tabla. También es posible modificar el comentario y el tipo de la tabla.

Si se usa **ALTER TABLE** para cambiar la especificación de una columna pero [DESCRIBE](#) tbl\_name indica que la columna no ha cambiado, es posible que MySQL haya ignorado la modificación por alguna razón. Por ejemplo, si se ha intentado cambiar una columna *VARCHAR* a *CHAR*, MySQL seguirá usando *VARCHAR* si la tabla contiene otras columnas de longitud variable.

**ALTER TABLE** trabaja haciendo una copia temporal de la tabla original. La modificación se realiza durante la copia, a continuación la tabla original se borra y la nueva se renombra. Esto se hace para

realizar que todas las actualizaciones se dirijan a la nueva tabla sin ningún fallo de actualización. Mientras **ALTER TABLE** se ejecuta, la tabla original permanece accesible en lectura para otros clientes. Las actualizaciones y escrituras en la tabla se retrasan hasta que la nueva tabla esté preparada.

Hay que tener en cuenta que si se usa otra opción para **ALTER TABLE** como *RENAME*, MySQL siempre creará una tabla temporal, aunque no sea estrictamente necesario copiarla (como cuando se cambia el nombre de una columna). Está previsto corregir esto en el futuro, pero como no es corriente usar **ALTER TABLE** para hacer esto, no es algo urgente de hacer. Para tablas **MyISAM**, se puede aumentar la velocidad de la recreación de índices (que es la parte más lenta del proceso) asignando un valor alto a la variable *myisam\_sort\_buffer\_size*.

- Para usar **ALTER TABLE**, es necesario tener los privilegios *ALTER*, *INSERT* y *CREATE* en la tabla.
- *IGNORE* es una extensión MySQL a SQL-92. Controla el modo de trabajar de **ALTER TABLE** si hay claves duplicadas o únicas en la nueva tabla. Si no se especifica *IGNORE*, la copia se aborta y se deshacen los cambios. Si se especifica *IGNORE*, en las filas duplicadas en una clave única sólo se copia la primera fila; el resto se eliminan.
- Se pueden usar múltiples cláusulas *ADD*, *ALTER*, *DROP* y *CHANGE* en una sentencia sencilla **ALTER TABLE**. Esto es una extensión MySQL a SQL-92, que permite sólo una aparición de cada cláusula en una sentencia **ALTER TABLE**.
- *CHANGE col\_name*, *DROP col\_name* y *DROP INDEX* también son extensiones MySQL a SQL-92.
- *MODIFY* es una extensión Oracle a **ALTER TABLE**.
- La palabra opcional *COLUMN* es una palabra ruidosa y puede ser omitida.
- Si se usa **ALTER TABLE tbl\_name RENAME TO new\_name** sin ninguna otra opción, MySQL sencillamente renombra los ficheros correspondientes a la tabla *tbl\_name*. No hay necesidad de crear una tabla temporal.
- Las cláusulas *create\_definition* usan la misma sintaxis para *ADD* y *CHANGE* que [CREATE TABLE](#). Esta sintaxis incluye sólo el nombre de columna, no el tipo.
- Se puede renombrar una columna usando una cláusula *CHANGE old\_col\_name create\_definition*. Para hacerlo, hay que especificar los nombres antiguo y nuevo de la columna y el tipo que la columna tiene actualmente. Por ejemplo, para renombrar una columna *INTEGER* desde a a b, se puede hacer esto:

```
mysql> ALTER TABLE t1 CHANGE a b INTEGER;
```

- Si se quiere cambiar el tipo de una columna, pero no su nombre, la sintaxis de *CHANGE* sigue necesitando un nombre de columna antiguo y nuevo, aunque mantenga en mismo nombre. Por ejemplo:

```
mysql> ALTER TABLE t1 CHANGE b b BIGINT NOT NULL;
```

- Sin embargo, desde la versión 3.22.16a de MySQL, se puede usar también *MODIFY* para modificar el tipo de una columna sin renombrarla:

```
mysql> ALTER TABLE t1 MODIFY b BIGINT NOT NULL;
```

- Si se usa *CHANGE* o *MODIFY* para acortar una columna para la cual existe un índice en parte de la columna (por ejemplo, si se tiene un índice en los primeros 10 caracteres de una columna *VARCHAR*), no será posible acortar la columna a un número de caracteres menos que los indexados.
- Cuando se cambia un tipo de columna usando *CHANGE* o *MODIFY*, MySQL intenta convertir datos al nuevo tipo lo mejor posible.
- En versiones 3.22 o siguientes de MySQL, se puede usar *FIRST* o *ADD ... AFTER col\_name* para añadir una columna en una posición específica dentro de una fila de la tabla. Por defecto se añade la columna al final. Desde MySQL 4.0.1, se pueden usar las palabras *FIRST* y *AFTER* en un *CHANGE* o *MODIFY*.
- *ALTER COLUMN* especifica un nuevo valor por defecto para una columna o elimina el valor por defecto anterior. Si se elimina el anterior valor por defecto y la columna puede ser *NULL*, el nuevo valor por defecto es *NULL*. Si la columna no puede ser *NULL*, MySQL asigna un nuevo valor por defecto, como se describe en [CREATE TABLE](#).
- *DROP INDEX* elimina un índice. Esto es una extensión MySQL para SQL-92. Ver la sintaxis de [DROP INDEX](#).
- Si se quitan columnas de una tabla, también se eliminan de cualquier índice de las que formen parte. Si todas las columnas que forman parte de un índice se eliminan, el índice se elimina también.
- Si la tabla contiene sólo una columna, ésta no puede ser eliminada. Si lo que se pretende es eliminar la tabla, se debe usar [DROP TABLE](#).
- *DROP PRIMARY KEY* elimina el índice primario. Si no existiera ese índice, se eliminará el primer índice *UNIQUE* de la tabla. (MySQL marca la primera clave *UNIQUE* como la *PRIMARY KEY* si no se ha especificado una *PRIMARY KEY* explícitamente.) Si se añade una *UNIQUE INDEX* o una *PRIMARY KEY* a la tabla, se almacena antes de cualquier índice *UNIQUE* de modo que MySQL pueda detectar claves duplicadas lo más pronto posible.
- *ORDER BY* permite crear una nueva tabla con un orden específico para las filas. La tabla no permanecerá en ese orden después de nuevas inserciones o borrados. En algunos casos, es más fácil hacer que MySQL ordene si la tabla está indexada por la columna por la que se desea ordenarla más tarde. Esta opción es corriente principalmente cuando se sabe que se va a consultar la tabla principalmente en un orden determinado; usando esta opción después de grandes campos en la tabla, es posible obtener un mejor rendimiento.
- Si se usa **ALTER TABLE** en una tabla **MyISAM**, todos los índices no únicos serán creados en

un proceso separado (como en [REPAIR](#)). Esto hace **ALTER TABLE** mucho más rápido cuando existe muchos índices.

- Desde MySQL 4.0 la característica anterior puede ser activada explícitamente. **ALTER TABLE ... DISABLE KEYS** hace que MySQL detenga la actualización de índices no únicos para tablas **MyISAM**. **ALTER TABLE ... ENABLE KEYS** debe ser usado para recrear índices perdidos. Como MySQL hace esto con un algoritmo especial que es mucho más rápido cuando se insertan claves una a una, desactivar las claves puede proporcionar una considerable mejora de tiempo cuando se inserta gran cantidad de filas.
- Con la función del API de C [mysql\\_info](#), se puede obtener cuantos registros han sido copiados, y (si se ha usado *IGNORE*) cuantos registros fueron borrados por la duplicación de valores de clave.
- Las cláusulas *FOREIGN KEY*, *CHECK* y *REFERENCES* actualmente no hacen nada, excepto para tablas del tipo **InnoDB** que soportan ... *ADD [CONSTRAINT [symbol]] FOREIGN KEY (...)* *REFERENCES ... (...)* y ... *DROP FOREIGN KEY ...*. Ver restricciones [FOREIGN KEY](#). La sintaxis para otros tipos de tabla se proporciona sólo por compatibilidad, para hacer más fácil portar código desde otros servidores SQL y para ejecutar aplicaciones que crean tablas con referencias.
- **ALTER TABLE** ignora las opciones de tabla *DATA DIRECTORY* y *INDEX DIRECTORY*.
- Si se quieren cambiar todas las columnas *CHAR/VARCHAR/TEXT* a un nuevo juego de caracteres (por ejemplo después de actualizar desde MySQL 4.0.x a 4.1.1) se puede hacer:

```
ALTER TABLE table_name CHARACTER SET character_set_name;
```

- El comando siguiente sólo cambia el juego de caracteres por defecto para la tabla:

```
ALTER TABLE table_name DEFAULT CHARACTER SET character_set_name;
```

- El juego de caracteres por defecto es el que se usa si no se especifica un juego de caracteres para una nueva columna que se añada a la tabla (por ejemplo con **ALTER TABLE ... ADD column**).

Seguidamente se muestra un ejemplo que demuestra como usar **ALTER TABLE**. Empezamos creando una tabla t1 como:

```
mysql> CREATE TABLE t1 (a INTEGER,b CHAR(10));
```

Para renombrar la tabla de t1 a t2:

```
mysql> ALTER TABLE t1 RENAME t2;
```

Para cambiar la columna a de *INTEGER* a *TINYINT NOT NULL* (dejando el mismo nombre), y cambiar la columna b de *CHAR(10)* a *CHAR(20)* y además renombrando de b a c:

```
mysql> ALTER TABLE t2 MODIFY a TINYINT NOT NULL, CHANGE b c CHAR(20);
```

Para añadir una columna *TIMESTAMP* llamada d:

```
mysql> ALTER TABLE t2 ADD d TIMESTAMP;
```

Para añadir un índice en la columna d, y hacer la columna a la clave primaria:

```
mysql> ALTER TABLE t2 ADD INDEX (d), ADD PRIMARY KEY (a);
```

Para eliminar la columna c:

```
mysql> ALTER TABLE t2 DROP COLUMN c;
```

Para añadir una nueva columna entera *AUTO\_INCREMENT* llamada c:

```
mysql> ALTER TABLE t2 ADD c INT UNSIGNED NOT NULL AUTO_INCREMENT,  
      ADD INDEX (c);
```

Anotar que debemos indexar en c, porque las columnas *AUTO\_INCREMENT* deben estar indexadas, y también que hemos declarado c como *NOT NULL*, porque las columnas indexadas no pueden ser *NULL*.

Cuando se añade una columna *AUTO\_INCREMENT*, los valores de columna se llenan con una secuencia numérica automáticamente. Se puede elegir el primer número de la secuencia ejecutando [SET INSERT\\_ID=value](#) antes de **ALTER TABLE** o usando la opción de tabla *AUTO\_INCREMENT=value*.

Con tablas **MyISAM**, si no se modifica la columna *AUTO\_INCREMENT*, la secuencia de numérica no resultará afectada. Si se elimina una columna *AUTO\_INCREMENT* y después se añade otra columna

*AUTO\_INCREMENT*, los números empezarán en 1 otra vez.

(4.0)

# ANALYZE TABLE

```
ANALYZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name[,tbl_name...]
```

Analiza y almacena la distribución de claves de una tabla. Durante el análisis, la tabla se bloquea para lectura. Funciona con tablas **MyISAM** y **BDB**.

Es equivalente a ejecutar *myisamchk -a* para la tabla.

MySQL usa la distribución de claves almacenada para decidir en qué orden pueden ser unidas las tablas cuando se hace una unión con algo distinto de una constante.

El comando devuelve una tabla con las siguientes columnas:

Columna	Valor
Table	Nombre de tabla
Op	Siempre <i>analyze</i> (análisis)
Msg_type	Uno de <i>status</i> , <i>error</i> , <i>info</i> o <i>warning</i> . (Estado, error, información o aviso)
Msg_text	El mensaje

Se puede verificar la distribución de claves almacenada con el comando [\*SHOW INDEX\*](#).

Si la tabla no ha cambiado desde el último comando **ANALYZE TABLE**, no será analizada de nuevo.

Antes de MySQL 4.1.1, los comandos **ANALYZE** no actualizaban el diario binario. Desde MySQL 4.1.1 lo hacen, salvo que se use la palabra clave opcional *NO\_WRITE\_TO\_BINLOG* (o su alias *LOCAL*).

(4.0)

# BACKUP TABLE

```
BACKUP TABLE tbl_name [, tbl_name] ... TO '/path/to/backup/directory'
```

**Nota:** esta sentencia está desaconsejada. Se está trabajando en algo mejor para reemplazarla que pueda proporcionar características de copia de seguridad en línea. En la actualidad, se puede usar el script *mysqlhotcopy* en su lugar.

**BACKUP TABLE** copia al directorio de copia de seguridad el número mínimo de ficheros de tablas necesarios para restaurar la tabla, después de escribir cualquier cambio al disco. La sentencia funciona sólo para tablas **MyISAM**. Copia los ficheros de definición '.frm' y el de datos '.MYD'. El fichero de índices '.MYI' puede ser reconstruido a partir de esos dos. El directorio debe ser especificado como un camino completo.

Durante la copia de seguridad, se hace un bloqueo de lectura para cada tabla, una en cada momento, hasta que se hayan copiado. Si se quiere hacer una copia de seguridad de varias tablas como una instantánea (impidiendo que cualquiera de ellas se modifique durante la operación de copia), primero se debe hacer un **LOCK TABLES** para obtener un bloqueo de lectura para todas las tablas del grupo.

La sentencia devuelve una tabla con las siguientes columnas:

Columna	Valor
Table	El nombre de la tabla
Op	Siempre 'backup'
Msg_type	Uno de los siguientes valores: status, error, info o warning (estado, error, información o aviso)
Msg_text	El mensaje

**BACKUP TABLE** está disponible a partir de MySQL 3.23.25.

(4.1.1)

# **START TRANSACTION**

## **COMMIT**

## **ROLLBACK**

Por defecto, MySQL se ejecuta en modo autocommit. Esto significa que tan pronto como se ejecuta una sentencia se actualiza (modifica) la tabla, MySQL almacenará la actualización en disco.

Si se están usando tablas de transacción segura (como **InnoDB** o **BDB**), se puede poner MySQL en modo no-autocommit con el comando siguiente:

```
SET AUTOCOMMIT=0
```

Después de desconectar el modo autocommit asignando cero a la variable *AUTOCOMMIT*, se debe usar **COMMIT** para almacenar los cambios en disco o **ROLLBACK** si se quieren ignorar los cambios hechos desde el principio de la transacción.

Si se quiere desactivar el modo autocommit para una serie de sentencias, se puede usar una sentencia **START TRANSACTION**:

```
START TRANSACTION;  
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;  
UPDATE table2 SET summmmary=@A WHERE type=1;  
COMMIT;
```

Se puede usar **BEGIN** y **BEGIN WORK** en lugar de **START TRANSACTION** para iniciar una transacción. **START TRANSACTION** fue añadido en MySQL 4.0.11; es la sintaxis SQL-99 y es el modo recomendado para empezar una transacción. **BEGIN** y **BEGIN WORK** están disponibles desde MySQL 3.23.17 y 3.23.19, respectivamente.

Si no se están usando tablas de transacción segura, cualquier cambio será almacenado inmediatamente, independientemente del estado del modo autocommit.

Si se usa una sentencia **ROLLBACK** después de actualizar una tabla no transaccional, se obtendrá un error (ER\_WARNING\_NOT\_COMPLETE\_ROLLBACK) como un aviso. Todas las tablas de transacción segura serán restauradas, pero cualquier tabla de transacción no segura no cambiará.

Si se usar **START TRANSACTION** o [\*SET AUTOCOMMIT=0\*](#), se debe usar el diario binario MySQL

para copias de seguridad en lugar del antiguo diario de actualización. Las transacciones se almacenan en el diario binario de una vez, después de **COMMIT**, para asegurar que las transacciones que se han rebobinado no se almacenen.

Se puede modificar el nivel de aislamiento para transacciones con *SET TRANSACTION ISOLATION LEVEL*.

(4.0)

# CHECK TABLE

```

CHECK TABLE tbl_name[,tbl_name...] [option [option...]]
option = QUICK | FAST | MEDIUM | EXTENDED | CHANGED

```

**CHECK TABLE** sólo funciona con tablas **MyISAM** e **InnoDB**. En tablas **MyISAM**, es lo mismo que ejecutar `myisamchk --medium-check table_name` en la tabla.

Si no se especifica una opción, se usa **MEDIUM**.

Verifica la tabla o tablas buscando errores. Para tablas **MyISAM**, se actualiza la clave de estadísticas. El comando devuelve una tabla con las siguientes columnas:

Columna	Valor
Table	Nombre de tabla
Op	Siempre <i>check</i> (verificado)
Msg_type	Uno de <i>status</i> , <i>error</i> , <i>info</i> o <i>warning</i> . (Estado, error, información o aviso)
Msg_text	El mensaje

Esta sentencia puede producir muchas filas de información para cada tabla verificada. La última fila será del tipo 'status' y normalmente será "OK". Si no se obtiene "OK", o la tabla está ya al día, normalmente se deberá ejecutar una reparación de la tabla. Que la tabla esté ya al día significa que el manipulador de almacenamiento para la tabla indicada no es necesario hacer una verificación.

Existen los siguientes tipos de verificación:

Tipo	Significado
QUICK	No recorre las filas para verificar enlaces incorrectos.
FAST	Sólo verifica tablas que no se hayan cerrado apropiadamente.
CHANGED	Sólo verifica tablas que hayan sido modificadas desde la última verificación o que no se hayan cerrado apropiadamente.
MEDIUM	Recorre las filas para verificar que los enlaces borrados son correctos. Esto también calcula el checksum de clave para las filas y lo compara con un checksum calculado para las claves.

**EXTENDED**

Realiza una comprobación completa de clave para todas las claves para cada fila. Esto asegura que la tabla es consistente al 100%, pero puede requerir mucho tiempo.

Para tablas **MyISAM** dimensionadas dinámicamente, una comprobación siempre será *MEDIUM*. Para filas dimensionadas estáticamente, se salta la fila para comprobaciones *QUICK* y *FAST* ya que esas filas raramente se corrompen.

Se pueden combinar opciones de comprobación, como en el siguiente ejemplo que realiza una comprobación rápida de la tabla para ver si fue cerrada apropiadamente:

```
CHECK TABLE test_table FAST QUICK;
```

Nota: en algunos casos **CHECK TABLE** puede modificar la tabla. Esto ocurre si la tabla está marcada como 'corrupta' o 'no cerrada apropiadamente' pero **CHECK TABLE** no encontró ningún problema en la tabla. En ese caso, **CHECK TABLE** marcará la tabla como correcta.

Si una tabla está corrupta, entonces es más probable que el problema esté en los índices y no en la parte de datos. Todos los tipos de comprobación anteriores verifican los índices minuciosamente y encuentran la mayor parte de los errores.

Si se quiere verificar una tabla que se asume que está bien, no se debe usar una opción o usar la opción *QUICK*. La segunda se debe usar cuando se está en un apuro y se puede correr un pequeño riesgo de que *QUICK* no encuentre un error en el fichero de datos. (En la mayoría de los casos MySQL encontrará, en un uso normal, cualquier error en el fichero de datos. Si eso ocurre la tabla se marcará como 'corrupta', y en ese caso no podrá ser usada hasta que sea reparada.)

*FAST* y *CHANGED* se usan más frecuentemente desde un script (por ejemplo para ser ejecutado desde **cron**) si se quiere verificar una tabla de vez en cuando. En la mayoría de los casos, *FAST* es preferible a *CHANGED*. (El único caso en que no lo es, es cuando se sospecha que se ha localizado un bug en el código de **MyISAM**.)

*EXTENDED* es solo para ser usado después de que se ha ejecutado una comprobación normal y aún se obtienen errores extraños desde una tabla cuando MySQL intenta actualizar o encontrar una fila por una clave (esto es muy probable si se ha realizado una comprobación normal).

Algunos problemas detectados por **CHECK TABLE** no pueden ser corregidos de forma automática:

- Si se encuentra una fila donde una columna `auto_increment` tiene valor 0. (Eso es posible asignando explícitamente 0 a la columna `AUTO_INCREMENT` al crear la fila con una sentencia [UPDATE](#).) Esto no es un error en sí mismo, pero puede causar problemas si se decide volcar la

tabla y restaurarla o hacer un ALTER TABLE en la tabla. En ese caso, la columna AUTO\_INCREMENT cambiará de valor, de acuerdo con las reglas de las columnas AUTO\_INCREMENT, lo cual puede causar problemas del tipo de error de clave duplicada. Para librarse del aviso bastará con ejecutar una sentencia UPDATE para asignar a la columna un valor distinto de cero.

(4.0)

# CHECKSUM TABLE

```
CHECKSUM TABLE tbl_name [, tbl_name] ... [ QUICK | EXTENDED ]
```

Devuelve la suma de comprobación (checksum) de una tabla.

Si se especifica *QUICK*, se devuelve el valor vivo de comprobación si está disponible, o NULL en caso contrario. Esto es muy rápido. El valor vivo se activa mediante la especificación de la opción de tabla *CHECKSUM=1*, que actualmente sólo está soportada en tablas **MyISAM**. Ver [CREATE TABLE](#).

En el modo *EXTENDED* se lee la tabla completa, línea a línea y se calcula el valor de comprobación. Esto puede ser muy lento para tablas grandes.

Por defecto, si no se especifica ni *QUICK* ni *EXTENDED*, MySQL devuelve el valor vivo si el motor de almacenamiento de tablas los soporta, y recorre la tabla en otro caso.

Esta sentencia está implementada a partir de MySQL 4.1.1.

(4.1.1)

# CREATE DATABASE

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
  [create_specification [, create_specification] ...]
create_specification:
  [DEFAULT] CHARACTER SET charset_name
  | [DEFAULT] COLLATE collation_name
```

**CREATE DATABASE** crea una base de datos con el nombre dado. Para usar **CREATE DATABASE** se necesita el privilegio **CREATE** en la base de datos.

Existen [reglas para los nombres](#) permitidos de bases de datos, tablas, índices, columnas.

Se producirá un error si la base de datos ya existe y no se ha especificado **IF NOT EXISTS**.

Desde MySQL 4.1.1, se pueden usar las opciones *create\_specification* para especificar características de las base de datos. Estas características se almacenan en el fichero 'db.opt' en el directorio de la base de datos. La cláusula *CHARACTER SET* especifica el conjunto de caracteres por defecto para la base de datos. La cláusula *COLLATE* especifica el conjunto de reglas de comparación de caracteres (collation) por defecto para la base de datos. Para más detalles sobre juegos de caracteres y reglas de comparación de caracteres ver [Caracteres y reglas](#).

En MySQL las bases de datos se implementan como directorios que contienen los ficheros correspondientes a las tablas de la base de datos. Como no hay tablas en una base de datos cuando esta es creada, la sentencia **CREATE DATABASE** sólo crea un directorio bajo el directorio "data" de MySQL (y el fichero 'db.opt' para MySQL 4.1.1 y siguientes).

*CREATE SCHEMA* se puede usar a partir de MySQL 5.0.2.

(4.1.1)

# CREATE INDEX

```
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name
  [USING index_type]
  ON tbl_name (index_col_name,...)
```

Sintaxis de `index_col_name`:

```
col_name [(length)] [ASC | DESC]
```

En MySQL 3.22 o posteriores, **CREATE INDEX** se interpreta como una sentencia [ALTER TABLE](#) para crear índices. La sentencia **CREATE INDEX** no hace nada antes de MySQL 3.22.

Normalmente, todos los índices de una tabla se crean al mismo tiempo que se crea la propia tabla con [CREATE TABLE](#). **CREATE INDEX** permite añadir índices a tablas existentes.

Una lista de columnas de la forma (col1,col2,...) crea un índice multi-columna. Los valores de índice se forman por la concatenación de valores de las columnas dadas.

Para columnas *CHAR* y *VARCHAR*, los índices pueden ser creados para usar sólo parte de una columna, usando la sintaxis `col_name(longitud)` para indexar un prefijo que contiene los primeros 'longitud' caracteres de cada valor de columna. Las columnas *BLOB* y *TEXT* también pueden ser indexada, pero se debe proporcionar una longitud de prefijo.

La sentencia mostrada a continuación crea un índice usando los 10 primeros caracteres de la columna 'name':

```
CREATE INDEX part_of_name ON customer (name(10));
```

Ya que la mayoría de los nombres normalmente se diferencian en los 10 primeros caracteres, el índice no debería ser mucho más lento que un índice creado con la columna 'name' completa. Además, usando una parte de la columna para los índices puede hacer el fichero de índice mucho más pequeño, con lo que se ahorra mucho espacio de disco y se mejora la velocidad en operaciones de inserción.

Los prefijos pueden tener hasta 255 bytes de longitud (o 1000 bytes para tablas **MyISAM** y **InnoDB** a partir de MySQL 4.1.2). Notar que estos límites de prefijos vienen dados en bytes, aunque la longitud

del prefijo en sentencias **CREATE INDEX** se interpreta como número de caracteres. Hay que tener esto en cuenta cuando se especifica una longitud de prefijo para una columna que usa un conjunto de caracteres multi-byte.

Sólo se puede añadir un índice en una columna que pueda contener valores *NULL* si se está usando la versión 3.23.2 o posterior de MySQL y si se usa una tabla de tipo **MyISAM**, **InnoDB** o **BDB**. Sólo se puede añadir un índice a una columna de tipo *BLOB* o *TEXT* si se está usando la versión 3.23.2 de MySQL o posterior y una tabla de tipo **MyISAM** o **BDB**, o MySQL 4.0.14 o posterior y una tabla de tipo **InnoDB**.

Una especificación de *index\_col\_name* puede terminar con *ASC* o *DESC*. Estas palabras clave están permitidas para extensiones futuras para especificar el almacenamiento de valores de índices ascendentes o descendentes. Actualmente se verifica su sintaxis, pero se ignoran; los valores de índices siempre se almacenan en orden ascendente.

A partir de MySQL 4.1.0, algunos motores de almacenamiento permiten especificar un tipo de índice cuando este es creado. La sintaxis para el especificador *index\_type* es *USING type\_name*. Los valores posibles de *type\_name* soportados por los diferentes motores de almacenamiento se muestran en la siguiente tabla. Cuando se listan varios tipos de índices, el primero es el valor por defecto cuando no se especifica uno.

Motor de almacenamiento	Tipos de índice disponibles
MyISAM	BTREE
InnoDB	BTREE
MEMORY/HEAP	HASH, BTREE

Ejemplo:

```
CREATE TABLE lookup (id INT) ENGINE = MEMORY;
CREATE INDEX id_index USING BTREE ON lookup (id);
```

Se puede usar *TYPE type\_name* como un sinónimo de *USING type\_name* para especificar un tipo de índice. Sin embargo, *USING* es el formato preferible. Además, el nombre de índice que precede al tipo de índice es la sintaxis de especificación de índice no es opcional con *TYPE*. Esto es así porque, al contrario que *USING*, *TYPE* no es una palabra reservada y puede ser interpretada como un nombre de índice.

Si se especifica un tipo de índice que no es legal para el motor de almacenamiento, pero existe otro tipo de índice disponible que el motor puede usar sin que afecte a los resultados de consultas, el motor podrá

usar el tipo disponible.

Los índices *FULLTEXT* sólo pueden indexar columnas *CHAR*, *VARCHAR* y *TEXT*, y sólo en tablas **MyISAM**. Los índices *FULLTEXT* están disponibles a partir de MySQL 3.23.23.

Los índices *SPATIAL* sólo pueden indexar columnas *spatial*, y sólo en tablas **MyISAM**. Los índices *SPATIAL* están disponibles a partir de MySQL 4.1.

(4.1.1)

# CREATE TABLE

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

O

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

Sintaxis de definición\_create:

```
definición_columnas
| [CONSTRAINT [símbolo]] PRIMARY KEY (index_nombre_col,...)
| KEY [nombre_index] (nombre_col_index,...)
| INDEX [nombre_index] (nombre_col_index,...)
| [CONSTRAINT [símbolo]] UNIQUE [INDEX]
  [nombre_index] [tipo_index] (nombre_col_index,...)
| [FULLTEXT|SPATIAL] [INDEX] [nombre_index] (nombre_col_index,...)
| [CONSTRAINT [símbolo]] FOREIGN KEY
  [nombre_index] (nombre_col_index,...) [definición_referencia]
| CHECK (expr)
```

Sintaxis de definición\_columnas:

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

Sintaxis de tipo:

```
TINYINT[(longitud)] [UNSIGNED] [ZEROFILL]
| SMALLINT[(longitud)] [UNSIGNED] [ZEROFILL]
| MEDIUMINT[(longitud)] [UNSIGNED] [ZEROFILL]
| INT[(longitud)] [UNSIGNED] [ZEROFILL]
```

## CREATE TABLE

```
INTEGER[(longitud)] [UNSIGNED] [ZEROFILL]
BIGINT[(longitud)] [UNSIGNED] [ZEROFILL]
REAL[(longitud,decimales)] [UNSIGNED] [ZEROFILL]
DOUBLE[(longitud,decimales)] [UNSIGNED] [ZEROFILL]
FLOAT[(longitud,decimales)] [UNSIGNED] [ZEROFILL]
DECIMAL(longitud,decimales) [UNSIGNED] [ZEROFILL]
NUMERIC(longitud,decimales) [UNSIGNED] [ZEROFILL]
DATE
TIME
TIMESTAMP
DATETIME
CHAR(longitud) [BINARY | ASCII | UNICODE]
VARCHAR(longitud) [BINARY]
TINYBLOB
BLOB
MEDIUMBLOB
LONGBLOB
TINYTEXT
TEXT
MEDIUMTEXT
LONGTEXT
ENUM(valor1,valor2,valor3,...)
SET(valor1,valor2,valor3,...)
tipo_spatial
```

### Sintaxis de nombre\_col\_index:

```
nombre_col [(longitud)] [ASC | DESC]
```

### Sintaxis de definición referencia:

```
REFERENCES nombre_tbl [(nombre_col_index,...)]
    [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
    [ON DELETE opción_referencia]
    [ON UPDATE opción_referencia]
```

### Sintaxis de opción\_referencia:

```
RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT
```

Sintaxis de opciones tabla:

```
opción_tabla [opción_tabla] ...
```

Sintaxis de opción\_tabla:

```
{ENGINE|TYPE} = {BDB|HEAP|ISAM|InnoDB|MERGE|MRG_MYISAM|MYISAM }
| AUTO_INCREMENT = valor
| AVG_ROW_LENGTH = valor
| CHECKSUM = {0 | 1}
| COMMENT = 'cadena'
| MAX_ROWS = valor
| MIN_ROWS = valor
| PACK_KEYS = {0 | 1 | DEFAULT}
| PASSWORD = 'cadena'
| DELAY_KEY_WRITE = {0 | 1}
| ROW_FORMAT = { DEFAULT|DYNAMIC|FIXED|COMPRESSED|REDUNTANT|COMPACT}
| RAID_TYPE = { 1 | STRIPED | RAID0 }
|     RAID_CHUNKS=valor
|     RAID_CHUNKSIZE=valor
| UNION = (nombre_tabla,[nombre_tabla...])
| INSERT_METHOD = { NO | FIRST | LAST }
| DATA DIRECTORY = 'camino de directorio absoluto'
| INDEX DIRECTORY = 'camino de directorio absoluto'
| [DEFAULT] CHARACTER SET nombre_conjunto_caracteres [COLLATE
nombre_cotejo]
```

Sintaxis de sentencia\_select:

```
[IGNORE | REPLACE] [AS] SELECT ...      (Alguna sentencia select legal)
```

**CREATE TABLE** crea una tabla con el nombre dado. Se debe poseer el privilegio *CREATE* para la tabla.

Las reglas para nombres válidos de tablas se pueden ver [aquí](#). Por defecto, la tabla se crea en la base de datos actual. Se producirá un error si la tabla ya existe, si no hay una base de datos actual o si la base de datos no existe.

En versiones de MySQL 3.22 y posteriores, el nombre de la tabla se puede especificar como db\_name.tbl\_name para la creación de la tabla en una base de datos específica. Esto funciona aunque no exista

una base de datos seleccionada. Si se escribe el nombre de la tabla entre comillas, el nombre de la base de datos y el de la tabla se deben entrecomillar separadamente. Por ejemplo, `midb`.`mitabla` es legal, pero `midb.mitabla` no lo es.

Desde la versión 3.23 de MySQL se puede usar la palabra clave *TEMPORARY* cuando se quiere crear una tabla. La tabla temporal sólo es visible para la conexión actual, y será borrada automáticamente cuando se cierre la conexión. Esto significa que dos conexiones diferentes pueden usar simultáneamente el mismo nombre para una tabla temporal sin conflictos entre ellas o con una tabla existente con el mismo nombre. (La tabla existente se ocultará hasta que la tabla temporal sea borrada). Desde MySQL 4.0.2 se debe tener el privilegio *CREATE TEMPORARY TABLES* para que sea posible crear tablas temporales.

Desde la versión 3.23 de MySQL se puede usar *IF NOT EXISTS* de modo que no se obtiene un error si la tabla ya existe. No se hace verificación de si la tabla existente tiene una estructura idéntica a la indicada por la sentencia *CREATE TABLE*.

Cada tabla *tbl\_name* se representa mediante algunos ficheros en el directorio de la base de datos. En el caso de tablas de tipo **MyISAM** se tendrá:

Fichero	Propósito
tbl_name.frm	Fichero de definición de formato de tabla
tbl_name.MYD	Fichero de datos
tbl_name.MYI	Fichero de índices

En la documentación de MySQL es posible encontrar información sobre el modo en que cada motor de almacenamiento crea los ficheros que representan las tablas.

Para detalles sobre tipos de columnas ver el [capítulo 5](#).

De momento no hemos incluido información sobre las extensiones espaciales de MySQL (spatial).

- Si no se especifica ni *NULL* ni *NOT NULL*, la columna se trata como si se hubiese especificado *NULL*.
- Una columna entera puede tener el atributo adicional *AUTO\_INCREMENT*. Cuando se inserta un valor *NULL* (que es lo recomendado) o 0 en una columna indexada *AUTO\_INCREMENT*, se usa como valor para la columna el siguiente valor secuencial. Normalmente ese valor es valor+1, donde valor es el mayor valor en la columna actual en la tabla. Las secuencias *AUTO\_INCREMENT* empiezan en 1. Ver [mysql\\_insert\\_id](#).

**Nota:** sólo puede existir una columna *AUTO\_INCREMENT* por tabla, debe estar indexada

y no puede tener un valor *DEFAULT*.

En la versión 3.23 de MySQL una columna *AUTO\_INCREMENT* sólo funcionará correctamente si contiene valores positivos. La inserción de un número negativo se considera como un número positivo muy grande. Esto se hace para evitar que por problemas de precisión números pasen de positivo a negativo y también para asegurar que accidentalmente una columna *AUTO\_INCREMENT* contenga un cero. En tablas **MyISAM** y **BDB** se puede especificar *AUTO\_INCREMENT* en una columna secundaria dentro de una clave multi-columna. Para hacer MySQL compatible con algunas aplicaciones ODBC, se puede encontrar el valor *AUTO\_INCREMENT* para la última fila insertada mediante la siguiente consulta:

```
SELECT * FROM tbl_name WHERE auto_col IS NULL
```

- Desde la versión 4.1 de MySQL, las definiciones de columnas de caracteres pueden incluir un atributo *CHARACTER SET* para especificar el conjunto de caracteres y, opcionalmente, un conjunto de reglas de comparación para la columna. Ver [apéndice C](#).

```
CREATE TABLE t (c CHAR(20) CHARACTER SET utf8 COLLATE utf8_bin);
```

También desde la versión 4.1, MySQL interpreta las especificaciones de longitud en definiciones de columnas de texto en caracteres. (Versiones anteriores las interpretaban en bytes.)

- Los valores *NULL* se manejan de modo diferente para columnas del tipo *TIMESTAMP* que para otros tipos. Antes de la versión 4.1.6 de MySQL no era posible almacenar un literal *NULL* en una columna *TIMESTAMP*; un valor *NULL* para la columna le asignaba el valor de fecha y hora actual. Debido a ese comportamiento de *TIMESTAMP*, los atributos *NULL* y *NOT NULL* no se aplicaban del modo normal, y eran ignorados si se especifican. Por otra parte, para hacer más fácil para los clientes MySQL el uso de columnas *TIMESTAMP*, el servidor informa que a esas columnas se les puede asignar valores *NULL*, lo cual es cierto, aunque *TIMESTAMP* nunca puede contener un valor *NULL*. Es posible ver esto cuando se usa [DESCRIBE](#) *tbl\_name* para obtener la descripción de una tabla. Por cierto, asignar 0 a una columna *TIMESTAMP* no es lo mismo que asignar *NULL*, ya que 0 es un valor válido para *TIMESTAMP*.
- La clausula *DEFAULT* especifica un valor por defecto para una columna. Con una excepción, el valor por defecto debe ser una constante, no puede ser una función o una expresión. Esto significa, por ejemplo, que no se puede asignar como fecha por defecto para una columna el valor de una función como [NOW\(\)](#) o [CURRENT\\_DATE](#). La excepción es que se puede asignar [CURRENT\\_TIMESTAMP](#) como el valor por defecto para una columna *TIMESTAMP* desde la versión 4.1.2 de MySQL. Si no se especifica un valor explícito por defecto para una columna, MySQL automáticamente asigna uno, como sigue. Si la columna puede tomar un valor *NULL*, la columna se define con una clausula *DEFAULT NULL* explícita. Si la columna no puede tomar el

valor *NULL*, MySQL define la columna con una clausula *DEFAULT* explícita, usando el valor implícito para el tipo de columna. El valor por defecto implícito se define como:

- Para tipos numéricos no declarados con el atributo *AUTO\_INCREMENT*, el valor por defecto es 0. Para las columnas *AUTO\_INCREMENT*, el valor por defecto es el siguiente valor dentro de la secuencia.
- Para tipos de fecha y tiempo distintos de *TIMESTAMP*, el valor por defecto es el valor cero apropiado para el tipo. Para la primera columna *TIMESTAMP* de la tabla, el valor por defecto es la fecha y hora actuales.
- Para tipos de cadena distintos de *ENUM*, el valor por defecto es la cadena vacía. Para los *ENUM*, el valor por defecto es el primero valor de la enumeración.

A las columnas *BLOB* y *TEXT* no se les puede asignar un valor por defecto.

- Se puede especificar un comentario para una columna con la opción *COMMENT*. El comentario se muestra mediante la sentencia [SHOW CREATE TABLE](#), y por [SHOW FULL COLUMNS](#). Esta opción está disponible desde MySQL 4.1. (Está permitida pero se ignora en versiones anteriores.)
- Desde la versión 4.1.0 de MySQL 4.1.0, se puede usar el atributo *SERIAL* como un alias para *BIGINT UNSIGNED NOT NULL AUTO\_INCREMENT UNIQUE*. Esta es una característica de compatibilidad.
- *KEY* normalmente es sinónimo de *INDEX*. Desde la versión 4.1, el atributo de clave *PRIMARY KEY* puede especificarse también como *KEY*. Esto se ha implementado para compatibilidad con otras base de datos.
- En MySQL, una clave *UNIQUE* sólo puede contener valores diferentes. Se produce un error si se intenta insertar una fila nueva con una clave que coincida con la de una fila existente. La excepción es si una columna en el índice puede tomar valores *NULL*, entonces puede contener múltiples valores *NULL*. Esta excepción no se aplica a tablas **BDB**, para las que una columna indexada permite sólo un valor *NULL*.
- Una *PRIMARY KEY* es una *KEY* única donde todas las columnas clave deben estar definidas como *NOT NULL*. Si no están declarados específicamente como *NOT NULL*, debe hacerse implícitamente (y discretamente). Una tabla sólo puede contener una *PRIMARY KEY*. Si no se tiene una y alguna aplicación solicita una, MySQL devolverá el primer índice *UNIQUE* que no tenga columnas *NULL*, como *PRIMARY KEY*.
- En la tabla creada, una *PRIMARY KEY* se coloca la primera, seguida por todos los índices *UNIQUE*, y después los índices no únicos. Esto ayuda al optimizador de MySQL para dar prioridad sobre qué índice usar y para detectar más rápidamente claves *UNIQUE* duplicadas.
- Una *PRIMARY KEY* puede ser un índice multicolumna. Sin embargo, no se puede crear un índice multicolumna usando el atributo *PRIMARY KEY* en la especificación de columna. Hacer esto marca sólo esa columna como primaria. Se debe usar una clausula adicional *PRIMARY KEY (nombre\_col\_index, ...)*.
- Si la clave *PRIMARY* o *UNIQUE* consiste sólo en una columna y es de tipo entero, se puede también referenciar como *\_rowid* en una sentencia [SELECT](#) (desde la Versión 3.23.11).
- En MySQL, el nombre de un *PRIMARY KEY* es *PRIMARY*. Para otros índices, si no se asigna un nombre, se le asigna el mismo nombre que la primera columna indexada, con un sufijo opcional (*\_2*, *\_3*. ...) para hacerlo único. Se pueden ver los nombres de los índices usando la sentencia

[SHOW INDEX FROM nombre\\_tabla.](#)

- Desde MySQL 4.1.0, algunos motores de almacenamiento permiten especificar un tipo de índice cuando este es creado. La sintaxis para el especificador de tipo\_índice es *USING nombre\_tipo*. Por ejemplo:

```
CREATE TABLE lookup
  (id INT, INDEX USING BTREE (id))
ENGINE = MEMORY;
```

Ver [CREATE INDEX](#) para más detalles.

- Sólo las tablas de tipos **MyISAM**, **InnoDB**, **BDB** y (desde MySQL 4.0.2) **MEMORY** soportan índices en columnas que contengan valores *NULL*. En otros casos se deben declarar esas columnas como *NOT NULL* o se producirá un error.
- Con la sintaxis *col\_name(longitud)* en una especificación de un índice, se puede crear un índice que use sólo los primeros 'longitud' bytes de una columna *CHAR* o *VARCHAR*. Esto puede hacer que el fichero de índices sea mucho más pequeño. Las tablas de tipos **MyISAM** y (desde MySQL 4.0.14) **InnoDB** soportan indexación en columnas *BLOB* y *TEXT*. Cuando se usa un índice en una columna *BLOB* o *TEXT* se debe especificar siempre la longitud de los índices. Por ejemplo:

```
CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));
```

Los prefijos pueden tener hasta 255 bytes de longitud (o 1000 bytes para tablas **MyISAM** y **InnoDB** desde MySQL 4.1.2). Notar que los límites de los prefijos se miden en bytes, aunque la longitud del prefijo en sentencias [CREATE INDEX](#) se interpretan como número de caracteres. Hay que tener esto en cuenta cuando se especifica una longitud de prefijo para una columna que usa un conjunto de caracteres multi-byte.

- Una especificación de nombre\_col\_índice puede terminar con *ASC* o *DESC*. Estas palabras clave están permitidas para futuras extensiones para la especificación de almacenamiento de índices ascendentes o descendentes. Actualmente se verifica su sintaxis, pero se ignoran; los valores de índices siempre se almacenan en orden ascendente.
- Cuando se usa *ORDER BY* o *GROUP BY* con una columna *TEXT* o *BLOB*, el servidor ordena los valores usando sólo los bytes iniciales indicados por la variable del servidor **max\_sort\_length**.
- En versiones de MySQL 3.23.23 o posteriores, se pueden crear índices especiales *FULLTEXT*. Se usan para búsquedas de texto completo. Sólo las tablas de tipo **MyISAM** soportan índices *FULLTEXT*. Sólo pueden ser creados desde columnas *CHAR*, *VARCHAR* y *TEXT*. La indexación siempre se hace sobre la columna completa; la indexación parcial no está soportada y cualquier longitud de prefijo es ignorada si se especifica.
- A partir de MySQL 4.1, se pueden crear índices *SPATIAL* en columnas de tipo espacial. Los tipos espaciales se soportan sólo para tablas **MyISAM** y las columnas indexadas deben declararse como *NOT NULL*.

- A partir de la versión 3.23.44 de MySQL, las tablas **InnoDB** soportan verificación para restricciones de claves foráneas. Hay que tener en cuenta que la sintaxis para *FOREIGN KEY* en **InnoDB** es mucho más restrictiva que la sintaxis presentada antes: las columnas en la tabla referenciada deben ser nombradas explícitamente. **InnoDB** soporta las acciones *ON DELETE* y *ON UPDATE* para claves ajenas tanto para MySQL 3.23.50 como para 4.0.8, respectivamente. Para otros tipos de tablas, el servidor MySQL verifica la sintaxis de *FOREIGN KEY*, *CHECK* y *REFERENCES* en comandos *CREATE TABLE*, pero no se toma ninguna acción.
- Para tablas **MyISAM** y **ISAM**, cada columna *NULL* requiere un bit extra, redondeando hacia arriba al siguiente byte. El tamaño máximo para un registro, en bytes, puede calcularse del siguiente modo:

```
row length = 1
             + (sum of column lengths)
             + (number of NULL columns + delete_flag + 7)/8
             + (number of variable-length columns)
```

El *delete\_flag* es 1 para tablas con formato de registro estático. Las tablas estáticas usan un bit en una fila de registro como un banderín que indica si la fila ha sido borrada.

*delete\_flag* es 0 para tablas dinámicas porque el banderín se almacena en una fila de cabecera dinámica. Estos cálculos no se aplican a tablas **InnoDB**, para las que el tamaño de almacenamiento no es diferente para columnas *NULL* en comparación con las *NOT NULL*.

La parte *opciones\_tabla* de **CREATE TABLE** sólo están disponibles desde MySQL 3.23.

Las opciones *ENGINE* y *TYPE* especifican el motor de almacenamiento para la tabla. *ENGINE* se añadió en MySQL 4.0.18 (para 4.0) y 4.1.2 (para 4.1). Esta es la opción aconsejada desde esas versiones, y *TYPE* queda desaconsejada. *TYPE* será soportada a lo largo de la serie 4.x series, pero será eliminada en MySQL 5.1.

Las opciones *ENGINE* y *TYPE* pueden tomar los valores siguientes:

Motor de almacenamiento	Descripción
BDB	Tablas de transacción segura con bloqueo de página.
BerkeleyDB	Alias para BDB.
HEAP	Los datos para esta tabla sólo se almacenan en memoria.
ISAM	El motor de almacenamiento original de MySQL.
InnoDB	Tablas de transacción segura con bloqueo de fila y claves foráneas.
MEMORY	Alias para HEAP.

MERGE	Una colección de tablas <b>MyISAM</b> usadas como una tabla.
MRG_MyISAM	Un alias para <b>MERGE</b> .
MyISAM	El nuevo motor binario de almacenamiento portable que reemplaza a <b>ISAM</b> .

Si se especifica un tipo de tabla, y ese tipo particular no está disponible, MySQL usará el tipo **MyISAM**. Por ejemplo, si la definición de la tabla incluye la opción *ENGINE=BDB* pero el servidor MySQL no soporta tablas **BDB**, la tabla se creará como una tabla **MyISAM**. Esto hace posible tener un sistema de réplica donde se tienen tablas operativas en el maestro pero las tablas creadas en el esclavo no son operativas (para obtener mayor velocidad). En MySQL 4.1.1 se obtiene un aviso si el tipo de tabla especificado no es aceptable.

Las otras opciones de tabla se usan para optimizar el comportamiento de la tabla. En la mayoría de los casos, no se tendrá que especificar ninguna de ellas. Las opciones trabajan con todos los tipos de tabla, salvo que se indique otra cosa:

Option	Descripción
AUTO_INCREMENT	El valor inicial <i>AUTO_INCREMENT</i> que se quiere seleccionar para la tabla. Sólo funciona en tablas <b>MyISAM</b> . Para poner el primer valor de un auto-incrementado para una tabla <b>InnoDB</b> , insertar una fila vacía con un valor una unidad menor, y después borrarla.
AVG_ROW_LENGTH	Una aproximación de la longitud media de fila de la tabla. Sólo se necesita esto para tablas largas con tamaño de registro variable. When you create a MyISAM table, MySQL uses the product of the MAX_ROWS and AVG_ROW_LENGTH options to decide how big the resulting table will be. If you don't specify either option, the maximum size for a table will be 4GB (or 2GB if your operating system only supports 2GB tables). The reason for this is just to keep down the pointer sizes to make the index smaller and faster if you don't really need big files. If you want all your tables to be able to grow above the 4GB limit and are willing to have your smaller tables slightly slower and larger than necessary, you may increase the default pointer size by setting the myisam_data_pointer_size system variable, which was added in MySQL 4.1.2.
CHECKSUM	Ponerlo a 1 si se quiere que MySQL mantenga un checksum para todas las filas (es decir, un checksum que MySQL actualiza automáticamente cuando la tabla cambia. Esto hace la tabla un poco más lenta al actualizar, pero hace más sencillo localizar tablas corruptas. La sentencia <b>CHECKSUM TABLE</b> devuelve el valor del checksum. (Sólo <b>MyISAM</b> ).
COMMENT	Un comentario de 60 caracteres para la tabla.

MAX_ROWS	Número máximo de filas que se planea almacenar en la tabla.
MIN_ROWS	Mínimo número de filas que se planea almacenar en la tabla.
PACK_KEYS	<p>Ponerlo a 1 si se quiere tener índices más pequeños. Esto normalmente hace que las actualizaciones sean más lentas y las lecturas más rápidas. Ponerlo a 0 desactiva cualquier empaquetado de claves. Ponerlo a <i>DEFAULT</i> (MySQL 4.0) indica al motor de almacenamiento que sólo empaquete columnas <i>CHAR/VARCHAR</i> largas. (Sólo <b>MyISAM</b> y <b>ISAM</b>). Si no se usa <i>PACK_KEYS</i>, por defecto sólo se empaquetan cadenas, no números. Si se usa <i>PACK_KEYS=1</i>, los números serán empaquetados también. Cuando se empaquetan claves de números binarios, MySQL usa compresión con prefijo:</p> <ul style="list-style-type: none"> <li>• Cada clave necesita un byte extra para indicar cuantos bytes de la clave anterior son iguales en la siguiente.</li> <li>• El puntero a la fila se almacena guardando primero el byte de mayor peso directamente después de la clave, para mejorar la compresión.</li> </ul> <p>Esto significa que si existen muchas claves iguales en dos filas consecutivas, todas las claves "iguales" que sigan generalmente sólo necesitarán dos bytes (incluyendo el puntero a la fila). Comparar esto con el caso corriente, donde las claves siguientes tendrán tamaño_almacenamiento_clave + tamaño_puntero (donde el tamaño del puntero normalmente es 4). Por otra parte, sólo se obtendrá un gran beneficio de la compresión prefija si hay muchos números que sean iguales. Si todas las claves son totalmente distintas, se usará un byte más por clave, si la clave no es una que pueda tener valores NULL. (En ese caso, la longitud de la clave empaquetada será almacenada en el mismo byte que se usa para marcar si la clave es NULL.)</p>
PASSWORD	Encripta el fichero <code>`.frm'</code> con un password. Esta opción no hace nada en la versión estándar de MySQL.
DELAY_KEY_WRITE	Poner esto a 1 si se quiere posponer la actualización de la tabla hasta que sea cerrada (sólo <b>MyISAM</b> ).
ROW_FORMAT	Define cómo deben almacenarse las filas. Actualmente esta opción sólo funciona con tablas <b>MyISAM</b> . El valor de la opción puede ser <i>DYNAMIC</i> o <i>FIXED</i> para formatos de fila estático o de longitud variable.

RAID_TYPE	<p>La opción <i>RAID_TYPE</i> permite exceder el límite de 2G/4G para un fichero de datos <b>MyISAM</b> (no el fichero de índice) en sistemas operativos que no soportan ficheros grandes. Obsérvese que esta opción es innecesaria y no se recomienda para sistemas de ficheros que soporten ficheros grandes. Se puede obtener mayor velocidad de cuellos de botella de entrada/salida usando directorios <i>RAID</i> en discos físicos diferentes. Por ahora, el único valor permitido para <i>RAID_TYPE</i> es <i>STRIPED</i>. 1 y <i>RAID0</i> son alias de <i>STREPED</i>. Si se especifica al opción <i>RAID_TYPE</i> para una tabla <b>MyISAM</b>, también es posible especificar las opciones <i>RAID_CHUNKS</i> y <i>RAID_CHUNKSIZE</i>. El valor máximo para <i>RAID_CHUNKS</i> es 255. <b>MyISAM</b> creará <i>RAID_CHUNKS</i> subdirectorios con los nombres '00', '01', '02',... '09', '0a', '0b,... en el directorio de la base de datos. En cada uno de esos directorios, <b>MyISAM</b> creará un "table_name.MYD". Cuando se escriban datos al fichero de datos, el manipulador <i>RAID</i> mapea los primeros <i>RAID_CHUNKSIZE</i> *1024 bytes al primer fichero, los siguientes <i>RAID_CHUNKSIZE</i> *1024 bytes al siguiente, y así sucesivamente. <i>RAID_TYPE</i> funciona en cualquier sistema operativo, siempre que se construya MySQL con la opción de configuración --with-raid. Para determinar si el servidor soporta tablas <i>RAID</i>, usar <a href="#">SHOW VARIABLES LIKE 'have_raid'</a> para ver si el valor de la variable es YES.</p>
UNION	<p><i>UNION</i> se usa cuando se quiere usar una colección de tablas idénticas como una. Esto sólo funciona con tablas <i>MERGE</i>. Por el momento es necesario tener los privilegios <i>SELECT</i>, <i>UPDATE</i> y <i>DELETE</i> en las tablas a mapear en una tabla <i>MERGE</i>. Originalmente, todas las tablas mapeadas deben pertenecer a la misma base de datos que la propia tabla <i>MERGE</i>. Esta restricción ha sido eliminada a partir de MySQL 4.1.1.</p>
INSERT_METHOD	<p>Si se quiere insertar tados en una tabla <i>MERGE</i>, se debe especificar con <i>INSERT_METHOD</i> dentro de qué la tabla se insertará la fila. <i>INSERT_METHOD</i> es una opción frecuente sólo en tablas <i>MERGE</i>. Esta opción se introdujo en MySQL 4.0.0.</p>
DATA DIRECTORY INDEX DIRECTORY	<p>Usando <i>DATA DIRECTORY='directory'</i> o <i>INDEX DIRECTORY='directory'</i> se puede especificar dónde colocará el motor de almacenamiento el fichero de datos y el de índices. Este directorio debe ser un camino completo al directorio (no un camino relativo). Esto sólo funciona con tablas <b>MyISAM</b> desde MySQL 4.0, cuando no se está usando la opción --skip-symlink.</p>

A partir de MySQL 3.23, se puede crear una tabla a partir de otra añadiendo la sentencia *SELECT* al final de la sentencia **CREATE TABLE**:

```
CREATE TABLE new_tbl SELECT * FROM orig_tbl;
```

MySQL creará nuevos campos para todos los elementos del **SELECT**. Por ejemplo:

```
mysql> CREATE TABLE test (a INT NOT NULL AUTO_INCREMENT,
->     PRIMARY KEY (a), KEY(b))
->     TYPE=MyISAM SELECT b,c FROM test2;
```

Esto creará una tabla **MyISAM** con tres columnas, a, b y c. Hay que tener en cuenta que esas tres columnas de la sentencia **SELECT** se añaden al lado derecho de la tabla, no superpuestos. Ver el siguiente ejemplo:

```
mysql> SELECT * FROM foo;
+----+
| n |
+----+
| 1 |
+----+
mysql> CREATE TABLE bar (m INT) SELECT n FROM foo;
Query OK, 1 row affected (0.02 sec)
Records: 1 Duplicates: 0 Warnings: 0

mysql> SELECT * FROM bar;
+-----+-----+
| m      | n |
+-----+-----+
| NULL  | 1 |
+-----+-----+
1 row in set (0.00 sec)
```

Para cada fila en la tabla *foo*, se inserta una fila en *bar* con los valores de *foo* y los valores por defecto para las nuevas columnas.

Si se produce cualquier error mientras se copian los datos a la tabla, esta se eliminará automáticamente y no se creará.

**CREATE TABLE ... SELECT** no creará índices de forma automática. Esto se ha hecho de forma intencionada para hacer el comando tan flexible como sea posible. Si se quiere tener índices en la tabla creada, se puede especificar después de la sentencias **SELECT**:

```
mysql> CREATE TABLE bar (UNIQUE (n)) SELECT n FROM foo;
```

Se pueden producir algunas conversiones de tipos de columna. Por ejemplo, el atributo *AUTO\_INCREMENT* no se preserva, y columnas *VARCHAR* se pueden convertir en *CHAR*.

Cuando se crea una tabla con **CREATE ... SELECT**, hay que asegurarse de crear alias para cualquier llamada a función o expresión en la consulta. Si no se hace, la sentencia *CREATE* podrá fallar o pueden resultar nombres de columna no deseados.

```
CREATE TABLE artists_and_works
SELECT artist.name, COUNT(work.artist_id) AS number_of_works
FROM artist LEFT JOIN work ON artist.id = work.artist_id
GROUP BY artist.id;
```

Desde MySQL 4.1, se puede especificar explícitamente el tipo para una columna generada:

```
CREATE TABLE foo (a TINYINT NOT NULL) SELECT b+1 AS a FROM bar;
```

En MySQL 4.1, también se puede usar *LIKE* para crear una tabla vacía basada en la definición de otra tabla, incluyendo cualquier atributo de columna e índices que tenga la tabla original:

```
CREATE TABLE new_tbl LIKE orig_tbl;
```

**CREATE TABLE ... LIKE** no copia ninguna opción de tabla *DATA DIRECTORY* o *INDEX DIRECTORY* que se haya especificado en la tabla original, o cualquier definición de clave foránea.

Se puede preceder el *SELECT* por *IGNORE* o *REPLACE* para indicar como manipular registros que dupliquen claves únicas. Con *IGNORE*, los nuevos registros que dupliquen la clave única de un registro existente serán descartados. Con *REPLACE*, los nuevos registros reemplazan a los que tengan el mismo valor de clave única. Si no se especifica *IGNORE* ni *REPLACE*, la repetición de claves únicas producirá un error.

Para asegurar que el diario de modificación puede ser usado para recrear las tablas originales, MySQL no permite la inserción concurrente durante *CREATE TABLE ... SELECT*.

## Cambios de especificaciones de columna sin notificación

En algunos casos, MySQL hace cambios en las especificaciones de columna dadas en una sentencia **CREATE TABLE** o **ALTER TABLE** sin notificarlo:

- Las columnas *VARCHAR* con una longitud menor de cuatro se cambian a *CHAR*.
- Si cualquier columna en una tabla tiene una longitud variable, la fila completa se convierte en una de longitud variable. Del mismo modo, si una tabla contiene alguna columna de longitud variable (*VARCHAR*, *TEXT* o *BLOB*), todas las columnas *CHAR* de más de cuatro caracteres se cambian a columnas *VARCHAR*. Esto no afecta al modo en que se usen estas columnas; en MySQL, *VARCHAR* es sólo una forma diferente de almacenar caracteres. MySQL realiza esta conversión porque esto ahorra espacio y hace las operaciones en la tabla más rápidas.
- A partir de MySQL 4.1.0, una columna *CHAR* o *VARCHAR* con una especificación de longitud mayor de 255 se convierte al tipo *TEXT* más pequeño que pueda contener valores de la longitud dada. Por ejemplo, *VARCHAR(500)* se convierte a *TEXT*, y *VARCHAR(200000)* se convierte a *MEDIUMTEXT*. Esto es un comportamiento para compatibilidad.
- Los tamaños de visualización de *TIMESTAMP* han sido descartados a partir de MySQL 4.1, debido a las modificaciones hechas al tipo de columna *TIMESTAMP* en esta versión. Antes de MySQL 4.1, los tamaños de visualización de *TIMESTAMP* deben aparecer siempre, y estar en el rango desde 2 a 14. Si se especifica un tamaño de visualización de 0 ó mayor que 14, el tamaño se ajusta a 14. Los valores impares en el rango entre 1 a 13 se ajustan al siguiente valor par.
- No se puede almacenar un valor literal *NULL* en una columna *TIMESTAMP*; asignar el valor *NULL* equivale a asignar el valor de fecha y hora actual. Debido a este comportamiento de las columnas *TIMESTAMP*, los atributos *NULL* y *NOT NULL* no se aplican de la forma normal y se ignoran si se especifican. [DESCRIBE tbl\\_name](#) siempre informa que a una columna *TIMESTAMP* se puede le pueden asignar valores *NULL*.
- Columns that are part of a PRIMARY KEY are made NOT NULL even if not declared that way.
- A partir de MySQL 3.23.51, los espacios iniciales son borrados automáticamente de las valores de los miembros *ENUM* y *SET* cuando la tabla es creada.
- MySQL mapea ciertos tipos de columna usados por otros productos de bases de datos SQL a los tipos de MySQL.
- Si se incluye una clausula *USING* para especificar un tipo índice que no es legal para un motor de almacenamiento, pero hay algún otro tipo de índice disponible que el motor puede usar sin afectar a los resultados de la consulta, el motor usará el tipo disponible.

Para comprobar si MySQL ha usado un tipo de columna diferente del especificado, se puede usar una sentencia [DESCRIBE](#) o [SHOW CREATE TABLE](#) después de crear la tabla o alterar una tabla.

Pueden producirse otros cambios de tipo de columna si se comprime una tabla usando *myisampack*.

(4.1.1)

# CREATE USER

```
CREATE USER user [IDENTIFIED BY [PASSWORD] 'password']  
[, user [IDENTIFIED BY [PASSWORD] 'password']] ...
```

La sentencia **CREATE USER** crea nuevas cuentas MySQL. 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 [GRANT](#).

La sentencia **CREATE USER** se añadió en MySQL 5.0.2.

(5.0.2)

# DELETE

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM table_name
      [WHERE where_definition]
      [ORDER BY ...]
      [LIMIT row_count]
```

Sintaxis multitabla:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
      table_name[*] [, table_name[*] ...]
FROM table-references
[WHERE where_definition]
```

O:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
      FROM table_name[*] [, table_name[*] ...]
      USING table-references
      [WHERE where_definition]
```

**DELETE** elimina columnas desde "table\_name" que satisfagan la condición dada por la "where\_definition", y devuelve el número de registros borrados.

Si se usa una sentencia **DELETE** sin cláusula *WHERE*, todas las filas serán borradas. Una forma más rápida de hacer esto, cuando no se necesita conocer el número de filas eliminadas, es usar [TRUNCATE TABLE](#).

En MySQL 3.23, **DELETE** sin la cláusula *WHERE* retorna cero como número de filas afectadas.

En esta versión, si realmente se quiere saber cuántas filas fueron eliminadas cuando se borran todas, y se está dispuesto a sufrir una pérdida de velocidad, se puede usar una sentencia **DELETE** con una cláusula *WHERE* que siempre se cumpla. Por ejemplo:

```
mysql> DELETE FROM table_name WHERE 1>0;
```

Esto es mucho más lento que **DELETE FROM table\_name** sin cláusula *WHERE*, porque borra filas una a una.

Si se borra la fila que contiene el valor máximo para una columna *AUTO\_INCREMENT*, ese valor podrá ser usado por una tabla **ISAM** o **BDB**, pero no por una tabla **MyISAM** o **InnoDB**. Si se borran todas las filas de una tabla con **DELETE FROM tbl\_name** (sin un *WHERE*) en modo *AUTO\_COMMIT*, la secuencia comenzará de nuevo para todos los motores de almacenamiento, excepto para **InnoDB** y (desde MySQL 4.0) **MyISAM**. Hay algunas excepciones a este comportamiento para tablas **InnoDB**.

Para tablas **MyISAM** y **BDB**, se puede especificar una columna secundaria *AUTO\_INCREMENT* en una clave multicolumna. En ese caso, la reutilización de los valores mayores de la secuencia borrados ocurre para tablas **MyISAM**.

La sentencia **DELETE** soporta los siguientes modificadores:

- Si se especifica la palabra *LOW\_PRIORITY*, la ejecución de **DELETE** se retrasa hasta que no existan clientes leyendo de la tabla.
- Para tablas **MyISAM**, si se especifica la palabra *QUICK* entonces el motor de almacenamiento no mezcla los permisos de índices durante el borrado, esto puede mejorar la velocidad en ciertos tipos de borrado.
- La opción *IGNORE* hace que MySQL ignore todos los errores durante el proceso de borrado. (Los errores encontrados durante en análisis de la sentencia se procesan normalmente.) Los errores que son ignorados por el uso de esta opción se devuelven como avisos. Esta opción apareció en la versión 4.1.1.

La velocidad de las operaciones de borrado pueden verse afectadas por otros factores, como el número de índices o el tamaño del caché para índices.

En tablas **MyISAM**, los registros borrados se mantienen en una lista enlazada y subsiguientes operaciones **INSERT** hacen uso de posiciones anteriores. Para recuperar el espacio sin usar y reducir el tamaño de los ficheros, usar la sentencia **OPTIMIZE TABLE** o la utilidad **myisamchk** para reorganizar tablas. **OPTIMIZE TABLE** es más fácil, pero **myisamchk** es más rápido.

El modificador *QUICK* afecta a si las páginas de índice se funden para operaciones de borrado. **DELETE QUICK** es más práctico para aplicaciones donde los valores de índice para las filas borradas serán reemplazadas por valores de índice similares para filas insertadas más adelante. En ese caso, los huecos dejados por los valores borrados serán reutilizados.

**DELETE QUICK** no es práctico cuando los valores borrados conducen a bloques de índices que no se llenan dejando huecos en valores de índice para nuevas inserciones. En ese caso, usar *QUICK* puede dejar espacios desperdiciados en el índice que permanecerán sin reclamar. Veamos un ejemplo de este

tipo de problema:

1. Crear una tabla que contiene un índice en una columna *AUTO\_INCREMENT*.
2. Insertar muchos registros en esa tabla. Cada inserción produce un valor de índice que es añadido al extremo mayor del índice.
3. Borrar un bloque de registros en el extremo inferior del rango de valores de la columna usando **DELETE QUICK**.

En este caso, los bloques de índices asociados con los valores de índice borrados quedan vacíos por debajo pero no se mezclan con otros bloques de índices debido al uso de *QUICK*. Esos bloques permanecen sin rellenar cuando se insertan nuevas filas, ya que los nuevos registros no tienen valores de índice en el rango borrado. Además, permanecerán sin rellenar aunque después se use **DELETE** sin *QUICK*, a no ser que alguno de los valores de índice borrados hagan que afecten a bloques de índices dentro o adyacentes a los bloques vacíos por debajo. Para liberar el espacio de índices no usado bajo estas circunstancias, se puede usar [OPTIMIZE TABLE](#).

Si se van a borrar muchas filas de una tabla, puede ser mucho más rápido usar **DELETE QUICK** seguido de [OPTIMIZE TABLE](#). Esto reconstruye el índice en lugar de reliazar muchas operaciones de mezcla de bloques de índice.

La opción *LIMIT row\_count*, específica de MySQL para **DELETE** indica al servidor el máximo número de filas a borrar antes de que el control se devuelva al cliente. Esto se puede usar para asegurar que una sentencia **DELETE** específica no tomará demasiado tiempo. Se puede repetir la sentencia **DELETE** hasta que el número de filas afectadas sea menor que el valor de *LIMIT*.

Si se usa una cláusula *ORDER BY* las filas serán borradas en el orden especificado por la cláusula. Esto sólo será práctico si se usa en conjunción con *LIMIT*. Por ejemplo, la siguiente sentencia encuentra filas que satisfagan la cláusula *WHERE*, las ordena por tiempos, y borra la primera (la más antigua):

```
DELETE FROM somelog
WHERE user = 'jcole'
ORDER BY timestamp
LIMIT 1
```

*ORDER BY* se puede usar con **DELETE** desde MySQL 4.0.0.

Desde MySQL 4.0, se pueden especificar múltiples tablas en la sentencia **DELETE** para eliminar filas de una o más tablas dependiendo de una condición particular en múltiples tablas. Sin embargo, no es posible usar *ORDER BY* o *LIMIT* en un **DELETE** multitabla.

La primera sintaxis de **DELETE** para tablas múltiples está soportada a partir de MySQL 4.0.0. La

segunda desde MySQL 4.0.2. La parte 'table\_references' lista las tablas involucradas en la fusión. Su sintaxis se describe en [JOIN](#).

Para la primera sintaxis, sólo se borran las filas coincidentes de las tablas listadas antes de la cláusula *FROM*. Para la segunda, sólo se borran las filas coincidentes de las tablas listadas en la cláusula *FROM* (antes de la cláusula *USING*). El efecto es que se pueden borrar filas de muchas tablas al mismo tiempo y además tener tablas adicionales que se usan para búsquedas:

```
DELETE t1,t2 FROM t1,t2,t3 WHERE t1.id=t2.id AND t2.id=t3.id
```

O

```
DELETE FROM t1,t2 USING t1,t2,t3 WHERE t1.id=t2.id AND t2.id=t3.id
```

Estas sentencias usan los tres ficheros cuando buscan filas para borrar, pero borran las filas coincidentes sólo de las tablas t1 y t2.

Los ejemplos muestran fusiones internas usando en operador coma, pero las sentencias **DELETE** multitabla pueden usar cualquier tipo de fusión (join) permitidas en sentencias [SELECT](#), como *LEFT JOIN*.

El .\* después de los nombres de tabla aparece sólo por compatibilidad con **Access**:

Si se usa una sentencia **DELETE** multitabla que afecte a tablas **InnoDB** para las que haya definiciones de claves foráneas, el optimizador MySQL procesará las tablas en un orden diferente del de la relación padre/hijo. En ese caso, la sentencia puede fallar y deshará los cambios (roll back). En su lugar, se debe borrar de una sola tabla y confiar en las capacidades de *ON DELETE* que proporciona **InnoDB** que harán que las otras tablas se modifiquen del modo adecuado.

**Nota:** en MySQL 4.0, se deben referir a los nombre de tablas a borrar mediante su verdadero nombre. En MySQL 4.1, se debe usar un alias (si se ha dado uno) cuando se haga referencia al nombre de la tabla:

En MySQL 4.0:

```
DELETE test FROM test AS t1, test2 WHERE ...
```

En MySQL 4.1:

```
DELETE t1 FROM test AS t1, test2 WHERE ...
```

El motivo por el que no se hizo este cambio en 4.0 es que no se quería romper cualquier aplicación anterior a 4.0 que usase la sintaxis antigua.

Actualmente, no se puede borrar de una tabla y seleccionar de la misma tabla en una subconsulta.

(4.1.1)

# DESCRIBE

```
{DESCRIBE | DESC} tbl_name [col_name | wild]
```

**DESCRIBE** es un sinónimo para [SHOW COLUMNS FROM](#).

**DESCRIBE** proporciona información sobre las columnas de una tabla. *col\_name* puede ser un nombre de columna o una cadena que contenga los caracteres comodín SQL '%' and '\_' para obtener salida sólo para las columnas cuyos nombres coincidan con la cadena. No es necesario escribir las cadenas entre comillas.

Si los tipos de las columnas son diferentes de lo que se esperaba basándose en la sentencia [CREATE TABLE](#) con la que se creó la tabla, hay que tener en cuenta que algunas veces MySQL cambia los tipos de las columnas.

Esta sentencia se proporciona por compatibilidad con Oracle.

La sentencia [SHOW](#) proporciona la misma información.

(4.0)

# DO

```
DO expression, [expression, ...]
```

**DO** ejecuta la expresión, pero no devuelve ningún resultado. Es una versión taquigráfica de una expresión SELECT, pero tiene la ventaja de que es un poco más rápida cuando el resultado no importa.

**DO** es más práctico cuando se usa con funciones que tienen efectos secundarios, como RELEASE LOCK.

**DO** se añadió en MySQL 3.23.47.

(4.1.1)

# DROP DATABASE

```
DROP {DATABASE | SCHEMA} [IF EXISTS] db_name
```

**DROP DATABASE** elimina todas las tablas de la base de datos y borra la base de datos. Hay que ser extremadamente cuidadoso con esta sentencia. Para usar **DROP DATABASE**, es necesario tener el privilegio *DROP* para la base de datos.

En MySQL 3.22 y posteriores, se pueden usar las palabras clave *IF EXISTS* para evitar el error que se produce si la base de datos no existe.

**DROP SCHEMA** se puede usar desde MySQL 5.0.2.

Si se usa **DROP DATABASE** en una base de datos enlazada simbólicamente, se eliminan tanto el enlace como la base de datos original.

A partir de MySQL 4.1.2, **DROP DATABASE** devuelve el número de tablas que fueron eliminadas. Este número corresponde a los ficheros '.frm' eliminados.

La sentencia **DROP DATABASE** elimina del directorio de la base de datos dada aquellos ficheros y directorios que haya creado el propio MySQL durante el funcionamiento normal:

- Todos los ficheros con estas extensiones:

.BAK	.DAT	.HSH	.ISD
.ISM	.ISM	.MRG	.MYD
.MYI	.db	.frm	

- Todos los subdirectorios con nombres que consistan en dos dígitos hexadecimales 00-ff. Estos son subdirectorios usados por tablas *RAID*.
- El fichero 'db.opt', si existe.

Si existen otros ficheros o directorios en el directorio de la base de datos después de que MySQL haya eliminado los listados antes, el directorio de la base de datos no podrá ser eliminado. En ese caso, se debe eliminar cualquier fichero o directorio que quede de forma manual y repetir la sentencia **DROP DATABASE** de nuevo.

También se pueden eliminar bases de datos usando *mysqladmin*.

(4.1.1)

DROP DATABASE

# ***DROP INDEX***

```
DROP INDEX index_name ON tbl_name
```

**DROP INDEX** elimina el índice con el nombre 'index\_name' de la tabla 'tbl\_name'. En MySQL 3.22 y posteriores, **DROP INDEX** se mapea como una sentencia [ALTER TABLE](#) para eliminar el índice. **DROP INDEX** no hace nada en versiones anteriores a MySQL 3.22.

(4.1.1)

# DROP TABLE

```
DROP [TEMPORARY] TABLE [IF EXISTS]
tbl_name [, tbl_name] ...
[RESTRICT | CASCADE]
```

**DROP TABLE** elimina una o más tablas. Se debe poseer el privilegio *DROP* para cada una de las tablas. Se eliminan tanto los datos que contengan y las definiciones de las tablas, así que hay que tener cuidado con esta sentencia.

En MySQL 3.22 y posteriores, se pueden usar las palabras clave *IF EXISTS* para evitar el error que ocurriría al intentar eliminar tablas que no existan. Desde MySQL 4.1, se genera una nota para cada tabla inexistente cuando se usa *IF EXISTS*. Ver [SHOW WARNINGS](#).

Se permite el uso de *RESTRICT* y *CASCADE* para hacer más sencilla la portabilidad. Pero por el momento, no hacen nada.

**Note:** **DROP TABLE** completa la transacción activa actual automáticamente, a no ser que se esté usando MySQL 4.1 o posterior y la palabra clave *TEMPORARY*.

La palabra clave *TEMPORARY* se ignora en MySQL 4.0. Desde la versión 4.1, tiene el efecto siguiente:

- La sentencia elimina sólo tablas temporales (*TEMPORARY*).
- La sentencia no termina ninguna transacción en curso.
- No se verifican derechos de acceso. (Una tabla temporal sólo es visible para el cliente que la ha creado, de modo que no es necesaria la verificación.)

Usar *TEMPORARY* es una buena forma de asegurar que no se eliminará una tabla no temporal de forma accidental.

(4.1.1)

# DROP USER

```
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 para [GRANT](#) o [REVOKE](#); por ejemplo, 'jeffrey'@'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* para la cuenta.

**DROP USER** se añadió en MySQL 4.1.1 y originalmente sólo borra 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.

Desde MySQL 5.0.2, se puede borrar una cuenta y sus privilegios mediante:

```
DROP USER user;
```

La sentencia borra los registros de privilegios para la cuenta de todas las tablas de concesiones.

Para MySQL entre 4.1.1 y 5.0.1, **DROP USER** sólo borra cuentas MySQL que no tengan ningún privilegio. Sirve para eliminar cada registro de cuenta de la tabla de usuarios. Para eliminar una cuenta MySQL, se puede usar el procedimiento siguiente, realizando cada paso en el orden mostrado:

Usar [SHOW GRANTS](#) para determinar qué privilegios tiene la cuenta.

Usar [REVOKE](#) para revocar los privilegios mostrados por [SHOW GRANTS](#). Esto borra los registros para la cuenta de todas las tablas de concesiones excepto la tabla *user*, y eliminar cualquier privilegio global listado en la tabla de usuario.

Borrar la cuenta usando **DROP USER** para eliminar el registro de la tabla de usuarios.

Antes de MySQL 4.1.1, **DROP USER** no está disponible. Primero se deben revocar los privilegios de la cuenta como se ha explicado. Después borrar el registro de la tabla *user* y restablecer las tablas de concesiones de este modo:

## DROP USER

```
mysql> DELETE FROM mysql.user  
      -> WHERE User='user_name' and Host='host_name';  
mysql> FLUSH PRIVILEGES;
```

(4.1.1)

# FLUSH

```
FLUSH [LOCAL | NO_WRITE_TO_BINLOG] flush_option [,flush_option] ...
```

Se puede usar el comando **FLUSH** si se desea limpiar algo del caché interno usado por MySQL. Para poder ejecutar **FLUSH**, se debe poseer el privilegio *RELOAD*.

El valor de *flush\_option* puede ser uno de los siguientes:

Opción	Descripción
HOSTS	Vacía el caché de tablas del host. Se pueden vaciar en disco las tablas del host si cambia de número IP o si se obtienen mensajes de error del tipo 'Host ... is blocked'. Cuando ocurren más de <i>max_connect_errors</i> errores en una fila para un host dado durante la conexión a un servidor MySQL, éste asume que algo anda mal y bloquea el host subsiguientes peticiones de conexión. Vaciar en disco las tablas del host le permite intentar conectar de nuevo. Se puede iniciar <b>mysqld</b> con <i>-O max_connect_errors=999999999</i> para evitar la aparición de este mensaje de error.
DES_KEY_FILE	Recarga las claves <i>DES</i> desde el fichero que se especificó con la opción <i>--des-key-file</i> durante el arranque del servidor.
LOGS	Cierra y reabre todos los ficheros de diario. Se se ha especificado un fichero de diario de actualización o un fichero de diario binario sin extensión, el número de extensión del fichero de diario será incrementado en una unidad con respecto al fichero anterior. Si se usa una extensión en el nombre de fichero, MySQL cerrará y reabrirá el fichero de diario de actualizaciones. Es lo mismo que enviar la señal <i>SIGHUP</i> al servidor <i>mysqld</i> .
PRIVILEGES	Recarga los privilegios desde las tablas de concesiones en la base de datos 'mysql'.
QUERY CACHE	Desfragmenta el caché de consulta para mejorar la utilización de su memoria. Este comando no eliminará ninguna consulta del caché, al contrario que <b>RESET QUERY CACHE</b> .

TABLES	Cierra todas las tablas abiertas y fuerza que todas las tablas en uso sean cerradas. Esto también vacía el caché de consultas.
[TABLE   TABLES] tbl_name [, tbl_name...]	Vacía en disco sólo las tablas dadas.
TABLES WITH READ LOCK	Cierra todas las tablas abiertas y bloquea todas las tablas para todas las bases de datos con un bloqueo de lectura hasta que se ejecute <a href="#">UNLOCK TABLES</a> . Esto proporciona una forma muy conveniente para hacer copias de seguridad si se posee un sistema de ficheros, como Veritas, que puede tomar instantáneas.
STATUS	Asigna a la mayoría de las variables de estado el valor cero. Esto es algo que sólo se debe hacer cuando se depura una consulta.
USER_RESOURCES	Asigna todos los recursos del usuario a cero. Esto permitirá bloquear usuarios para que entren de nuevo.

Antes de MySQL 4.1.1, los comandos **FLUSH** no actualizaban el diario binario. Desde MySQL 4.1.1 lo hacen salvo que se use la palabra clave *NO\_WRITE\_TO\_BINLOG* (o su alias *LOCAL*), o salvo que el comando contenga uno de estos argumentos: *LOGS*, *MASTER*, *SLAVE*, *TABLES WITH READ LOCK*, porque cualquiera de esos argumentos puede causar problemas si es preplicado a un esclavo.

También se puede acceder a algunos de los comandos mostrados arriba con la utilidad *mysqladmin*, usando *flush-hosts*, *flush-logs*, *flush-privileges*, *flush-status* o *flush-tables*.

Se recomienda echar un vistazo al comando [RESET](#) usado con replicación.

(4.0)

# GRANT

## REVOKE

```
GRANT priv_type [(column_list)] [, priv_type [(column_list)]] ...
  ON {tbl_name | * | *.* | db_name.*}
  TO user [IDENTIFIED BY [PASSWORD] 'password']
      [, user [IDENTIFIED BY [PASSWORD] 'password']] ...
  [REQUIRE
    NONE |
    [{SSL| X509}]
    [CIPHER 'cipher' [AND]]
    [ISSUER 'issuer' [AND]]
    [SUBJECT 'subject']]
  [WITH [GRANT OPTION | MAX_QUERIES_PER_HOUR count |
        MAX_UPDATES_PER_HOUR count |
        MAX_CONNECTIONS_PER_HOUR count |
        MAX_USER_CONNECTIONS count]]
```

```
REVOKE priv_type [(column_list)] [, priv_type [(column_list)]] ...
  ON {tbl_name | * | *.* | db_name.*}
  FROM user [, user] ...
```

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM user [, user] ...
```

Las sentencias **GRANT** y **REVOKE** permiten a los administradores del sistema crear cuentas de usuario MySQL y conceder y revocar derechos de esas cuentas. **GRANT** y **REVOKE** están disponibles a partir de MySQL 3.22.11. Para versiones anteriores de MySQL, estas sentencias no hacen nada.

La información sobre cuentas MySQL se almacena en las tablas de la base de datos *mysql*. Esta base de datos y el control de acceso se describen en detalle en la sección 5 "Database Administration", que se puede consultar para detalles adicionales.

Los privilegios pueden ser concedidos en varios niveles:

### Nivel global

Los privilegios globales se aplican a todas las bases de datos de un servidor dado. Estos privilegios se

almacenan en la tabla *mysql.user*. **GRANT ALL ON \*.\*** y **REVOKE ALL ON \*.\*** conceden y revocan sólo privilegios globales.

## Nivel de base de datos

Los privilegios de base de datos se aplican a todos los objetos en una base de datos dada. Estos privilegios se almacenan en las tablas *mysql.db* y *mysql.host*. **GRANT ALL ON db\_name.\*** y **REVOKE ALL ON db\_name.\*** conceden y revocan sólo privilegios de base de datos.

## Nivel de tabla

Los privilegios de tabla se aplican a todas las columnas de una tabla dada. Estos privilegios se almacenan en la tabla *mysql.tables\_priv*. **GRANT ALL ON db\_name.tbl\_name** y **REVOKE ALL ON db\_name.tbl\_name** conceden y revocan únicamente privilegios de tabla.

## Nivel de columna

Los privilegios de columna se aplican a una columna individual en una tabla dada. Estos privilegios se almacenan en la tabla *mysql.columns\_priv*. Cuando se usa **REVOKE**, se deben especificar las mismas columnas que cuando se concedieron los privilegios.

## Nivel de rutina

Los privilegios *CREATE ROUTINE*, *ALTER ROUTINE*, *EXECUTE* y *GRANT* se aplican a rutinas almacenadas. Pueden ser concedidos en los niveles global y de base de datos. Además, excepto para *CREATE ROUTINE*, estos privilegios pueden ser concedidos en el nivel de rutina para rutinas individuales y se almacenan en la tabla *mysql.procs\_priv*.

Para hacer más sencillo revocar todos los privilegios, MySQL 4.1.2 ha añadido la siguiente sintaxis, que elimina todos los privilegios de los niveles de global, de base de datos, tabla y columna para los usuarios nombrados:

```
mysql> REVOKE ALL PRIVILEGES, GRANT OPTION FROM user [, user] ...
```

Antes de la versión 4.1.2 de MySQL, no es posible eliminar todos los privilegios de una vez. Se necesitan dos sentencias:

```
mysql> REVOKE ALL PRIVILEGES ON *.* FROM user [, user] ...  
mysql> REVOKE GRANT OPTION ON *.* FROM user [, user] ...
```

Para las sentencias **GRANT** y **REVOKE**, se puede usar cualquiera de los siguientes valores para *priv\_type*:

Privilegio	Significado
ALL [PRIVILEGES]	Activa todos los privilegios excepto <i>GRANT OPTION</i> .
ALTER	Permite el uso de <a href="#">ALTER TABLE</a> .
CREATE	Permite el uso de <a href="#">CREATE TABLE</a> .
CREATE ROUTINE	Crear rutinas almacenadas.
CREATE TEMPORARY TABLES	Permite el uso de <a href="#">CREATE TEMPORARY TABLE</a> .
CREATE VIEW	Permite el uso de <a href="#">CREATE VIEW</a> .
DELETE	Permite el uso de <a href="#">DELETE</a> .
DROP	Permite el uso de <a href="#">DROP TABLE</a> .
EXECUTE	Permite al usuario ejecutar procedimientos almacenados.
FILE	Permite el uso de <a href="#">SELECT ... INTO OUTFILE</a> y <a href="#">LOAD DATA INFILE</a> .
INDEX	Permite el uso de <a href="#">CREATE INDEX</a> y <a href="#">DROP INDEX</a> .
INSERT	Permite el uso de <a href="#">INSERT</a> .
LOCK TABLES	Permite el uso de <a href="#">LOCK TABLES</a> en tablas sobre las que ya se posea el privilegio <i>SELECT</i> .
PROCESS	Permite el uso de <a href="#">SHOW FULL PROCESSLIST</a> .
REFERENCES	No implementado.
RELOAD	Permite el uso de <a href="#">FLUSH</a> .
REPLICATION CLIENT	Permite al usuario preguntar dónde están los servidores esclavo o maestro.
REPLICATION SLAVE	Necesario para la replicación esclava (para leer eventos del diario binario desde el maestro).
SELECT	Permite el uso de <a href="#">SELECT</a> .
SHOW DATABASES	La sentencia <a href="#">SHOW DATABASES</a> muestra todas las bases de datos.
SHOW VIEW	Permite el uso de <a href="#">SHOW CREATE VIEW</a> .
SHUTDOWN	Permite el uso del apagado de <i>mysqladmin shutdown</i> .

SUPER	Permite el uso de las sentencias <a href="#">CHANGE MASTER</a> , <a href="#">KILL</a> , <a href="#">PURGE MASTER LOGS</a> y <a href="#">SET GLOBAL</a> , el comando depurador de <i>mysqladmin debug</i> ; permite conectar (una vez) aunque se haya alcanzado el número de conexiones <i>max_connections</i> .
UPDATE	Permite el uso de <a href="#">UPDATE</a> .
USAGE	Sinónimo de "sin privilegios".
GRANT OPTION	Permite conceder privilegios.

Los privilegios *CREATE TEMPORARY TABLES*, *EXECUTE*, *LOCK TABLES*, *REPLICATION CLIENT*, *REPLICATION SLAVE*, *SHOW DATABASES* y *SUPER* fueron añadidos en MySQL 4.0.2. (*EXECUTE* no estará activo hasta MySQL 5.0.3.) *CREATE VIEW* y *SHOW VIEW* se añadirán en MySQL 5.0.1. *CREATE ROUTINE* y *ALTER ROUTINE* en MySQL 5.0.3. Para usar estos privilegios cuando se actualice a partir de una versión anterior de MySQL que no los tenga, se deben actualizar las tablas de concesiones.

El privilegio *REFERENCES* no se usa actualmente.

En versiones más antiguas de MySQL que no tienen el privilegio *SUPER*, hay que especificar el privilegio *PROCESS* en su lugar.

*USAGE* se puede usar cuando se quiere crear un usuario que no tenga privilegios.

Usar [SHOW GRANTS](#) para determinar qué privilegios tiene una cuenta.

Se pueden asignar privilegios globales mediante el uso de la sintaxis *ON \*.\** o privilegios de base de datos mediante la sintaxis *ON db\_name.\**. Si se especifica *ON \** y se tiene seleccionada una base de datos por defecto, los privilegios se conceden a esa base de datos. (Cuidado: si se especifica *ON \** y no se tiene ninguna base de datos seleccionada por defecto, los privilegios se conceden globalmente.)

Los privilegios *EXECUTION*, *FILE*, *PROCESS*, *RELOAD*, *REPLICATION CLIENT*, *REPLICATION SLAVE*, *SHOW DATABASES*, *SHUTDOWN* y *SUPER* son privilegios administrativos que sólo pueden ser concedidos de forma global (usando la sintaxis *ON \*.\**).

Otros privilegios pueden ser concedidos globalmente o en niveles más específicos.

Los únicos valores *priv\_type* que se pueden especificar para una tabla son *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *CREATE*, *DROP*, *GRANT OPTION*, *INDEX* y *ALTER*.

Los únicos valores *priv\_type* que se pueden especificar para una columna (esto es, cuando se usa una

cláusula *column\_list*) son *SELECT*, *INSERT* y *UPDATE*.

Los únicos valores *priv\_type* que se pueden especificar en el nivel de rutina son *ALTER ROUTINE*, *EXECUTE* y *GRANT*. *CREATE ROUTINE* no es un privilegio de nivel de rutina porque se debe tener este privilegio para que sea posible crear una rutina en primer lugar.

Para los niveles global, de base de datos, tabla y rutina, **GRANT ALL** sólo asigna privilegios que existan en el nivel en que se están concediendo. Por ejemplo, si se usa **GRANT ALL ON db\_name.\***, que es una sentencia de nivel de base de datos, ningún privilegio exclusivo del nivel global, como *FILE* será concedido.

Para privilegios de nivel de columna (esto es, cuando se especifica *column\_list*), se deben nombrar los privilegios a conceder explícitamente. No se puede usar *ALL* como especificador de privilegio.

MySQL permite conceder privilegios de nivel de base de datos aunque la base de datos no exista, para hacer más sencilla la preparación de una base de datos para su uso. Sin embargo, MySQL no permite actualmente conceder privilegios a nivel de tabla o columna si la tabla no existe. De forma similar, no se pueden conceder privilegios de nivel de rutina a procedimientos que no existan.

MySQL no revoca ningún privilegio automáticamente aunque se elimine una tabla o una base de datos. Si se elimina una rutina y se han concedido privilegios de nivel de rutina para ella, estos serán revocados.

**Nota:** los comodines '\_' y '%' están permitidos cuando se especifican nombres de bases de datos en sentencias **GRANT** que concedan privilegios en los niveles global o de base de datos. Esto significa, por ejemplo, que si se quiere usar un carácter '\_' como parte de un nombre de base de datos, se debe especificar como '\\_' en la sentencia **GRANT**, para prevenir que el usuario al que se están concediendo pueda acceder a otras bases de datos que coincidan con el patrón definido por el comodín; por ejemplo, **GRANT ... ON `foo\\_bar`.\* TO ...**

Para albergar derechos concedidos a usuarios para máquinas arbitrarias, MySQL soporta la especificación para el valor de usuario en el formato *user\_name@host\_name*. Si un nombre de usuario *user\_name* o de máquina *host\_name* es un valor legal como identificador sin entrecomillar, no será necesario entrecomillarlo. Sin embargo, las comillas serán necesarias para especificar una cadena *user\_name* que contenga caracteres especiales (como '-'), o una cadena de *host\_name* que contenga caracteres especiales o caracteres comodín (como '%'); por ejemplo, 'test-user'@'test-hostname'. Hay que entrecomillar el nombre de usuario y el de la máquina separadamente.

Se pueden especificar comodines en el nombre de máquina. Por ejemplo, *user\_name@%.loc.gov* se aplica a *user\_name* para cualquier máquina en el dominio *loc.gov* y *user\_name@'144.155.166.%'* se aplica a *user\_name* para cualquier máquina en la clase C subred *144.155.166*.

El formato simplificado *user\_name* es sinónimo de *user\_name@'%'*.

MySQL no soporta comodines en nombres de usuario. Los usuarios anónimos se definen mediante la inserción de entradas con *User=""* en la tabla *mysql.user* o creando un usuario con un nombre vacío con la sentencia **GRANT**:

```
mysql> GRANT ALL ON test.* TO ''@'localhost' ...
```

Cuando se especifican valores entre comillas para base de datos, tablas, columnas y nombres de rutina como identificadores, se usan tildes a izquierda (`). Los nombres de máquina, de usuario y contraseñas como cadenas, se usan apóstrofes (").

**Aviso:** si se permite la conexión al servidor MySQL de usuarios anónimos, también se deben conceder privilegios a todos los usuarios locales como *user\_name@localhost*. En caso contrario, se usará la cuenta del usuario anónimo para la máquina local en la tabla *mysql.user* cuando usuarios con nombre intenten conectarse al servidor MySQL desde la máquina local. (Esta cuenta de usuario anónimo se crea durante la instalación de MySQL.)

Se puede determinar si esto se aplica a cada caso mediante la ejecución de la siguiente sentencia:

```
mysql> SELECT Host, User FROM mysql.user WHERE User='';
```

Si se quiere borrar la cuenta de usuario anónimo local para evitar el problema descrito, usar estas sentencias:

```
mysql> DELETE FROM mysql.user WHERE Host='localhost' AND User='';
mysql> FLUSH PRIVILEGES;
```

**GRANT** soporta nombres de máquina de hasta 60 caracteres. Los nombres de base de datos, tabla, columna y rutina pueden tener hasta 64 caracteres. Los nombres de usuario hasta 16 caracteres.

Los privilegios para una tabla o columna se forman aditivamente con el O lógico de los privilegios de cada uno de los cuatro niveles de privilegios. Por ejemplo, si la tabla *mysql.user* especifica que un usuario tiene un privilegio global *SELECT*, el privilegio no puede ser denegado por una entrada en el nivel de base de datos, tabla o columna.

Los privilegios para una columna se pueden calcular de este modo:

```

global privileges
OR (database privileges AND host privileges)
OR table privileges
OR column privileges

```

En la mayoría de los casos, sólo se conceden privilegios a un usuario en uno de los niveles de privilegio, de modo que la vida no suele tener esta complicación.

Si se conceden privilegios para una combinación de usuario/máquina que no existe en la tabla *mysql.user*, se añade una entrada y se conserva hasta que se elimina con una sentencia [DELETE](#). En otras palabras, **GRANT** puede crear entradas en la tabla de usuarios, pero **REVOKE** no las eliminará; se debe hacer explícitamente usando [DROP USER](#) o [DELETE](#).

A partir de MySQL 3.22.12, si se crea un nuevo usuario o si se posee privilegios globales, la contraseña del usuario se asigna con la contraseña especificada por la cláusula *IDENTIFIED BY*, si se proporciona una. Si el usuario tiene una contraseña, se reemplaza con la nueva.

**Aviso:** si se crea un nuevo usuario pero no se especifica una cláusula *IDENTIFIED BY*, el usuario no tiene contraseña. Esto es inseguro. A partir de MySQL 5.0.2, se puede activar el modo SQL *NO\_AUTO\_CREATE\_USER* para evitar que **GRANT** cree nuevos usuarios si de otro modo los haría, a no ser que se proporcione una contraseña en *IDENTIFIED BY*.

Las contraseñas también se pueden asignar mediante la sentencia [SET PASSWORD](#).

En la cláusula *IDENTIFIED BY*, la contraseña puede ser proporcionada como un valor de contraseña literal. No es necesario usar la función [PASSWORD\(\)](#) como sí lo es para la sentencia [SET PASSWORD](#). Por ejemplo:

```
GRANT ... IDENTIFIED BY 'mypass' ;
```

Si no se quiere enviar la contraseña en texto claro y se conoce el valor que se devuelve por la función [PASSWORD\(\)](#) para la contraseña, se puede especificar tal valor precedido por la palabra clave *PASSWORD*:

```
GRANT ... IDENTIFIED BY PASSWORD
 '*6C8989366EAF75BB670AD8EA7A7FC1176A95CEF4' ;
```

En un programa C, se puede obtener ese valor mediante el uso de la función del API C

[make\\_scrambled\\_password\(\)](#).

Si se conceden privilegios para una base de datos, se crea una entrada en la tabla *mysql.db* si es necesario. Si todos los privilegios para una base de datos se eliminan con **REVOKE**, esa entrada será eliminada.

Si un usuario no tiene privilegios para una tabla, el nombre de la tabla no se muestra cuando el usuario pide una lista de tablas (por ejemplo, con una sentencia [SHOW TABLES](#)).

El privilegio *SHOW DATABASES* permite a una cuenta ver los nombres de bases de datos mediante la sentencia [SHOW DATABASES](#). Las cuentas que no tienen este privilegio sólo ven las bases de datos para las que tienen algún privilegio, y no pueden usar la sentencia en ningún caso si el servidor fue arrancado con la opción *--skip-show-database*.

La cláusula *WITH GRANT OPTION* proporciona al usuario la oportunidad de dar a otros usuarios cualquier privilegio que éste tenga en el nivel de privilegios especificado. Se debe ser cuidadoso con a quien se le da el privilegio *GRANT OPTION*, porque dos usuarios con privilegios diferentes pueden unirlos.

No se puede conceder a otro usuario un privilegio que no se posee; el privilegio *GRANT OPTION* permite dar sólo aquellos privilegios que se poseen.

Hay que saber que cuando se concede a un usuario el privilegio *GRANT OPTION* en un nivel particular, cualquier privilegio que posea el usuario (o que se le conceda en el futuro) en ese nivel, también se puede conceder por ese usuario. Dupongamos que se concede a un usuario el privilegio *INSERT* en una base de datos. Si se concede a continuación el privilegio *SELECT* en la base de datos y se especifica *WITH GRANT OPTION*, el usuario puede dar a otros no sólo el privilegio *SELECT*, sino también el *INSERT*. Si a continuación se concede el privilegio *UPDATE* al usuario en la base de datos, puede conceder a otros los privilegios *INSERT*, *SELECT* y *UPDATE*.

No se deben conceder privilegios *ALTER* a un usuario normal. Si se hace, el usuario puede intentar trastornar el sistema de privilegios mediante el renombrado de tablas.

Las opciones *MAX\_QUERIES\_PER\_HOUR count*, *MAX\_UPDATES\_PER\_HOUR count* y *MAX\_CONNECTIONS\_PER\_HOUR count* son nuevas en MySQL 4.0.2. Sirven para limitar el número de consultas, actualizaciones y entradas que un usuario puede realizar durante una hora. Si el contador se pone a 0 (el valor por defecto), significa que no hay límite para ese usuario.

La opción *MAX\_USER\_CONNECTIONS count* es nueva en MySQL 5.0.3. Limita el número máximo de conexiones simultáneas que puede hacer la cuenta. Si *count* es 0 (valor por defecto), la variable de sistema *max\_user\_connections* determina el número de conexiones simultáneas para la cuenta.

**Nota:** para especificar cualquiera de estas opciones de limitación de recursos para un usuario sin que se afecten los privilegios ya existentes, usar **GRANT USAGE ON \*.\* ... WITH MAX\_...**

MySQL puede verificar los atributos de certificado **X509** además de la autenticación habitual que se basa en nombre de usuario y contraseña. Para especificar opciones relacionadas con **SSL** para una cuenta MySQL, usar la cláusula *REQUIRE* para las sentencias **GRANT**.

Existen distintas posibilidades para limitar tipos de conexión para una cuenta:

- Si una cuenta no tiene los requerimientos **SSL** o **X509**, se permiten conexiones sin encriptar si el nombre de usuario y contraseña son válidos. Sin embargo, las conexiones encriptadas también se pueden usar en la opción del cliente, si éste tiene el certificado apropiado y los ficheros de clave.
- La opción *REQUIRE SSL* dice al servidor que permita sólo conexiones encriptadas **SSL** para la cuenta. Hay que tener en cuenta que esta opción puede ser omitida si existe cualquier registro de control de acceso que permita conexiones no **SSL**.

```
mysql> GRANT ALL PRIVILEGES ON test.* TO 'root'@'localhost'
-> IDENTIFIED BY 'goodsecret' REQUIRE SSL;
```

- *REQUIRE X509* significa que el cliente debe tener un certificado válido pero que el certificado exacto, emisor y condición no importan. El único requisito es que debe ser posible verificar su firma con uno de los certificados **CA**.

```
mysql> GRANT ALL PRIVILEGES ON test.* TO 'root'@'localhost'
-> IDENTIFIED BY 'goodsecret' REQUIRE X509;
```

- *REQUIRE ISSUER 'issuer'* pone la restricción en los intentos de conexión de que el cliente debe presentar un certificado **X509** válido emitido por el **CA** 'issuer'. Si el cliente presenta un certificado que es válido pero tiene un emisor diferente, el servidor rechaza la conexión. El uso de certificados **X509** siempre implica encriptado, de modo que la opción **SSL** es innecesaria.

```
mysql> GRANT ALL PRIVILEGES ON test.* TO 'root'@'localhost'
-> IDENTIFIED BY 'goodsecret'
-> REQUIRE ISSUER '/C=FI/ST=Some-State/L=Helsinki/
O=MySQL Finland AB/CN=Tonu Samuel/Email=tonu@example.com';
```

Notar que el valor *ISSUER* debe ser introducido como una única cadena.

- *REQUIRE SUBJECT 'subject'* pone otra restricción en los intentos de conexión, la de que el

cliente debe presentar un certificado válido **X509** con la condición 'subject' en él. Si el cliente presenta un certificado válido pero que tiene una condición diferente, el servidor rechaza la conexión.

```
mysql> GRANT ALL PRIVILEGES ON test.* TO 'root'@'localhost'
-> IDENTIFIED BY 'goodsecret'
-> REQUIRE SUBJECT '/C=EE/ST=Some-State/L=Tallinn/
O=MySQL demo client certificate/
CN=Tonu Samuel/Email=tonu@example.com';
```

Notar que el valor *SUBJECT* debe ser introducido como una única cadena.

- *REQUIRE CIPHER 'cipher'* es necesario para asegurar que se usa un cifrado lo bastante fuerte y claves lo suficientemente largas. **SSL** mismo puede ser débil si se usan algoritmos antiguos con claves de encriptado cortas. Usando esta opción, se puede preguntar por un método de cifrado concreto para permitir la conexión.

```
mysql> GRANT ALL PRIVILEGES ON test.* TO 'root'@'localhost'
-> IDENTIFIED BY 'goodsecret'
-> REQUIRE CIPHER 'EDH-RSA-DES-CBC3-SHA';
```

Las opciones *SUBJECT*, *ISSUER* y *CIPHER* pueden combinarse en la cláusula *REQUIRE* así:

```
mysql> GRANT ALL PRIVILEGES ON test.* TO 'root'@'localhost'
-> IDENTIFIED BY 'goodsecret'
-> REQUIRE SUBJECT '/C=EE/ST=Some-State/L=Tallinn/
O=MySQL demo client certificate/
CN=Tonu Samuel/Email=tonu@example.com'
-> AND ISSUER '/C=FI/ST=Some-State/L=Helsinki/
O=MySQL Finland AB/CN=Tonu Samuel/Email=tonu@example.com'
-> AND CIPHER 'EDH-RSA-DES-CBC3-SHA';
```

Notar que cada valor *SUBJECT* e *ISSUER* deben ser introducidos como una única cadena.

A partir de MySQL 4.0.4, la palabra clave *AND* es opcional entre las opciones *REQUIRE*.

El orden de las opciones no es importante, pero ninguna opción se puede especificar dos veces.

Cuando *mysqld* arranca, todos los privilegios se leen en memoria. Los privilegios de base de datos, tabla

y columna tienen efecto en seguida, y los privilegios de nivel de usuario tienen efecto la vez siguiente que el usuario se conecte. Las modificaciones en las tablas de concesiones que se realicen usando **GRANT** o **REVOKE** se notifican al servidor inmediatamente. Si se modifican las tablas de concesiones manualmente (usando **INSERT**, **UPDATE**, etc), se debe ejecutar una sentencia **FLUSH PRIVILEGES** o ejecutar `mysqladmin flush-privileges` para indicar al servidor que recargue las tablas de concesiones.

Si se están usando privilegios de tabla o columna para un único usuario, el servidor examina los privilegios de tabla y columna para todos los usuarios y esto ralentiza MySQL un poco. De forma similar, si se limita el número de consultas, actualizaciones o conexiones para algunos usuarios, el servidor debe monitorizar esos valores.

Las mayores diferencias entre las versiones de **GRANT** de SQL estándar y de MySQL son:

- En MySQL, los privilegios están asociados con una combinación de usuario/contraseña y no sólo con un usuario.
- SQL estándar no tiene privilegios de nivel global o de base de datos, ni soporta todos los tipos de privilegio que soporta MySQL.
- MySQL no soporta los privilegios de SQL estándar *TRIGGER* o *UNDER*.
- Los privilegios de SQL estándar están estructurados de forma jerárquica. Si se elimina un usuario, todos los privilegios que tiene concedidos se revocan. Esto también es cierto en MySQL 5.0.2 y siguientes si se usa **DROP USER**. Antes de 5.0.2, los privilegios concedidos no son revocados automáticamente; deben ser revocados directamente.
- Con SQL estándar, cuando se elimina una tabla, todos los privilegios para la tabla son revocados. Con SQL estándar, cuando se revoca un privilegio, todos los privilegios concedidos basados en él son revocados también. En MySQL, los privilegios pueden ser eliminados sólo con la sentencia **REVOKE** explícita o mediante la manipulación de las tablas de concesiones de MySQL.
- En MySQL, si sólo se posee el privilegio *INSERT* en algunas de las columnas de una tabla, se pueden ejecutar sentencias **INSERT** en la tabla; las columnas para las que no se posee el privilegio *INSERT* serán asignadas a sus valores por defecto. SQL estándar requiere que se posea el privilegio *INSERT* en todas las columnas.

(4.1.1)

# HANDLER

```

HANDLER tbl_name OPEN [ AS alias ]
HANDLER tbl_name READ index_name { = | >= | <= | < } (value1,value2,...)
    [ WHERE ... ] [LIMIT ... ]
HANDLER tbl_name READ index_name { FIRST | NEXT | PREV | LAST }
    [ WHERE ... ] [LIMIT ... ]
HANDLER tbl_name READ { FIRST | NEXT }
    [ WHERE ... ] [LIMIT ... ]
HANDLER tbl_name CLOSE

```

La sentencia **HANDLER** proporciona un acceso directo al interfaz del motor de almacenamiento de una tabla **MyISAM**.

El primer formato de la sentencia **HANDLER**, permite el acceso mediante subsiguientes sentencias **HANDLER ... READ**. Este objeto tabla no se comparte con otros hilos y no podrá cerrarse hasta que el hilo llame a **HANDLER tbl\_name CLOSE** o hasta que el hilo termine.

El segundo formato recoge una fila (o más, especificadas mediante la cláusula *LIMIT*) donde el índice especificado satisfaga los valores dados y la condición *WHERE* se cumpla. Si se tiene un índice multicolumna, se deben especificar los valores de índice de columna como una lista separada con comas. Una de dos: o se especifican los valores para todas las columnas de un índice, o valores que definan un prefijo de las columnas más a la izquierda del índice. Supongamos un índice que incluya tres columnas llamadas col\_a, col\_b y col\_c, en ese orden. La sentencia **HANDLER** puede especificar valores para las tres columnas en el índice, o para las columnas más a la izquierda. Por ejemplo:

```

HANDLER ... index_name = (col_a_val,col_b_val,col_c_val) ...
HANDLER ... index_name = (col_a_val,col_b_val) ...
HANDLER ... index_name = (col_a_val) ...

```

El tercer formato recoge una fila (o más, especificadas mediante la cláusula *LIMIT*) de la tabla según el orden del índice, y según la condición *WHERE*.

El cuarto formato (sin especificación de índice) recoge una fila (o más, especificadas mediante la cláusula *LIMIT*) de la tabla en el orden natural de las filas (tal como está almacenadas en el fichero de datos) que cumplan la condición *WHERE*. Es más rápido que **HANDLER tbl\_name READ index\_name** cuando se desea hacer un recorrido completo de la tabla.

**HANDLER ... CLOSE** cierra una tabla que fue abierta con **HANDLER ... OPEN**.

Nota: Si se usa el interfaz **HANDLER** para una *PRIMARY KEY* se debe recordar entrecomillar la palabra *PRIMARY* con acentos a la izquierda: **HANDLER tbl READ `PRIMARY` > (...)**.

**HANDLER** es algo como una sentencia de bajo nivel. Por ejemplo, no proporciona consistencia. Es decir, **HANDLER ... OPEN** no toma una copia instantanea de la tabla, y no bloquea la tabla. Esto significa que después de hacer un **HANDLER ... OPEN**, los datos de la tabla pueden ser modificados (por este o por otro hilo) y esas modificaciones pueden aparecer sólo de forma parcial en recorridos **HANDLER ... NEXT** o **HANDLER ... PREV**.

Las razones para usar este interfaz en lugar de SQL normal son:

- Es más rápido que **SELECT** porque:
  - Se asigna un motor de almacenamiento para el hilo declarado en **HANDLER OPEN**.
  - Hay menos análisis involucrado.
  - No hay optimización y comprobación de consultas.
  - La tabla usada no tiene que ser bloqueada entre dos peticiones *handler*.
  - EL interfaz no tiene que proporcionar un aspecto consistente de los datos (por ejemplo, se permiten "lecturas sucias"), de modo que el motor de almacenamiento puede hacer optimizaciones que SQL normalmente no puede hacer.
- Hace mucho más fácil portar aplicaciones que usen un **ISAM** como interfaz para MySQL.
- Permite tratar una base de datos de un modo que no es fácil (en algunos casos imposible) hacer con SQL. El interfaz es un modo más natural de ver datos cuando se trabaja con aplicaciones que proporcionan un interfaz de usuario interactivo para la base de datos.

(4.0)

# INSERT

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
  [INTO] tbl_name [(col_name,...)]
  VALUES ({expression | DEFAULT},...),(...),...
  [ ON DUPLICATE KEY UPDATE col_name=expression, ... ]
```

O

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
  [INTO] tbl_name
  SET col_name={expression | DEFAULT}, ...
  [ ON DUPLICATE KEY UPDATE col_name=expression, ... ]
```

O

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
  [INTO] tbl_name [(col_name,...)]
  SELECT ...
```

**INSERT** inserta nuevas filas en una tabla existente. Los formatos *INSERT ... VALUES* e *INSERT ... SET*, insertas filas basándose en los valores especificados explícitamente. El formato *The INSERT ... SELECT* inserta filas seleccionadas de otra tabla o tablas. El formato *INSERT ... VALUES* con una lista de múltiples valores está soportada por MySQL desde la versión 3.22.5. La sintaxis *INSERT ... SET* está soportada por MySQL desde la versión 3.22.10.

*tbl\_name* es la tabla donde se insertarán las filas. Las columnas para las que la sentencia proporciona valores se pueden especificar de las siguientes formas:

- Las lista de nombres de columnas o la clausula *SET* indican las columnas explícitamente.
- Si no se especifica una lista de columnas para *INSERT ... VALUES* o *INSERT ... SELECT*, se deben proporcionar valores para todas las columnas en la lista *VALUES()* o por *SELECT*. Si no se conoce el orden de las columnas dentro de la tabla, usar [\*DESCRIBE tbl\\_name\*](#) para encontrarlo.

Los valores de columnas se pueden proporcionar de varias formas:

- Si se está ejecutando MySQL en el modo estricto (strict mode), a cualquier columna para la que

no se proporcione un valor explícitamente se le asignará su valor por defecto (explícito o implícito). Por ejemplo, si se especifica una lista de columnas que no contiene el nombre de todas las columnas de la tabla, a las columnas sin nombre le serán asignados sus valores por defecto. La asignación de valores por defecto se describe en [CREATE TABLE](#). Si se quiere que las sentencias **INSERT** generen un error a no ser que se especifiquen falores de forma explícita para todas las columnas que no tengan un valor por defecto, se debe usar el modo *STRICT*.

- Se puede usar la palabra clave *DEFAULT* para poner una columna a su valor por defecto de forma explícita. (Nuevo en MySQL 4.0.3.) Esto hace más sencillo escribir sentencias **INSERT** que asignan valores a casi todas las columnas, porque permite la escritura de una lista *VALUES* incompleta, que no incluye un valor para cada columna de la tabla. En otro caso, se debería escribir la lista de nombres de columnas correspondiente a cada valor en la lista *VALUES*. A partir de MySQL 4.1.0, se puede usar *DEFAULT(col\_name)* como una forma más general que puede ser usada en expresiones para obtener valores por defecto de columnas.
- Si tanto la lista de columnas como la de *VALUES* están vacías, **INSERT** crea una fila en la que cada columna tendrá su valor por defecto:

```
mysql> INSERT INTO tbl_name () VALUES();
```

- Se puede especificar una expresión *expr* para proporcionar un valor de columna. Esto forzará complejas conversiones de tipo si el de la expresión no coincide con el tipo de la columna, y la conversión de un valor dado puede provocar diferentes valores insertados dependiendo del tipo de la columna. Por ejemplo, insertar la cadena '1999.0e-2' en una columna INT, FLOAT, DECIMAL(10,6) o YEAR producirá los valores 1999, 19.9921, 19.992100 y 1999. El motivo es que el valor almacenado en una columna INT y YEAR sea 1999 es que la conversión de cadena a entero mira sólo la parte inicial de la cadena que se pueda considerar un valor entero o un año válido. Para columnas en punto flotante o punto fijo, la conversión de cadena a punto flotante tiene en cuenta la cadena completa como un valor válido en coma flotante. Una expresión *expr* se puede referir a cualquier columna que se haya asignado previamente en la lista de valores. Por ejemplo, se puede hacer esto, ya que el valor de col2 se refiere a col1, que ya ha sido asignado:

```
mysql> INSERT INTO tbl_name (col1,col2) VALUES(15,col1*2);
```

Pero no se puede hacer esto, porque el valor para col1 se refiere a col2, que se asigna después de col1:

```
mysql> INSERT INTO tbl_name (col1,col2) VALUES(col2*2,15);
```

La excepción es para columnas que contengan valores autoincrementados. Esto es debido a que

el valor *AUTO\_INCREMENT* se genera después de la asignación de cualquier otro valor, así que cualquier referencia a una columna *AUTO\_INCREMENT* devolverá un 0.

La sentencia **INSERT** soporta los modificadores siguientes:

- Si se especifica la palabra clave *DELAYED*, el servidor coloca la fila o filas a insertar en un búfer, y el cliente lanza la sentencia **INSERT DELAYED** y puede continuar. Si la tabla está ocupada, el servidor guarda las filas. Cuando la tabla queda libre, empieza a insertar filas, verificando periódicamente para ver si hay nuevas peticiones de lectura para la tabla. Si las hay, la cola de inserción retardada se suspende hasta que la tabla quede libre de nuevo. *DELAYED* se añadió en MySQL 3.22.5.
- Si se especifica la palabra clave *LOW\_PRIORITY*, la ejecución de la sentencia **INSERT** se retrasa hasta que no haya clientes leyendo de la tabla. Esto incluye a otros clientes que empiecen a leer mientras existan clientes leyendo, y mientras la sentencia **INSERT LOW\_PRIORITY** está esperando. Por lo tanto, es posible que un cliente que lance una sentencia **INSERT LOW\_PRIORITY** permanezca esperando por mucho tiempo (eventualmente para siempre) en un entorno con muchas lecturas. (Esto contrasta con lo que sucede con **INSERT DELAYED**, que permite al cliente continuar inmediatamente). *LOW\_PRIORITY* no debe ser usado con tablas **MyISAM** que no permiten inserciones concurrentes.
- Si se especifica la palabra clave *HIGH\_PRIORITY*, se anula el efecto de la opción *--low-priority-updates* si el servidor fue arrancado con esa opción. Esto también hace que no se puedan usar inserciones concurrentes. *HIGH\_PRIORITY* fue añadido en MySQL 3.23.11.
- El valor de filas afectadas por una sentencia **INSERT** se puede obtener usando la función del API C [mysql\\_affected\\_rows\(\)](#).
- Si se especifica la palabra *IGNORE* en una sentencia **INSERT**, los errores que se produzcan mientras se ejecuta la sentencia se tratan con avisos. Por ejemplo, sin *IGNORE* una fila que duplique un valor de clave *PRIMARY* o *UNIQUE* existente en la tabla provocará un error y la sentencia será abortada. Con *IGNORE*, el error se ignora y la fila no será insertada. Las conversiones de datos que produzcan errores abortarán la sentencia si no se especifica *IGNORE*. Con *IGNORE*, los valores inválidos se ajustarán al valor más próximos y se insertarán; se producirán avisos pero la sentencia no se aborta. Se puede determinar cuantas filas fueron insertadas mediante la función del API [mysql\\_info](#).

Si se especifica la cláusula *ON DUPLICATE KEY UPDATE* (nueva en MySQL 4.1.0), y se inserta una fila que puede provocar un valor duplicado en una clave *PRIMARY* o *UNIQUE*, se realiza un *UPDATE* (actualización) de la fila antigua. Por ejemplo, si se declara una columna 'a' como *UNIQUE* y ya contiene el valor 1, las dos sentencias siguientes tienen el mismo efecto:

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3)
-> ON DUPLICATE KEY UPDATE c=c+1;
```

```
mysql> UPDATE table SET c=c+1 WHERE a=1;
```

El valor de filas afectadas es 1 si la fila es insertada como un nuevo registro y 2 si se actualiza un registro ya existente.

**Nota:** si la columna 'b' es única también, la sentencia **INSERT** puede ser equivalente a esta sentencia **UPDATE**:

```
mysql> UPDATE table SET c=c+1 WHERE a=1 OR b=2 LIMIT 1;
```

Si  $a=1$  OR  $b=2$  se cumple para varias filas, sólo una será actualizada. En general, se debe intentar evitar el uso de la cláusula *ON DUPLICATE KEY* en tablas con múltiples claves *UNIQUE*.

Desde MySQL 4.1.1 es posible usar la función *VALUES(col\_name)* en una cláusula *UPDATE* para referirse a los valores de columna en la parte *INSERT* de una sentencia *INSERT ... UPDATE*. En otras palabras, *VALUES(col\_name)* en una cláusula *UPDATE* se refiere al valor *col\_name* que será insertado si no existe un conflicto de clave duplicada. Esta función es especialmente corriente en inserciones de varias filas. La función *VALUES* sólo tiene sentido en sentencias *INSERT ... UPDATE* y devuelve *NULL* en otro caso.

Ejemplo:

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3),(4,5,6)
-> ON DUPLICATE KEY UPDATE c=VALUES(a)+VALUES(b);
```

El comando anterior es idéntico a las dos sentencias siguientes:

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3)
-> ON DUPLICATE KEY UPDATE c=3;
mysql> INSERT INTO table (a,b,c) VALUES (4,5,6)
-> ON DUPLICATE KEY UPDATE c=9;
```

Cuando se usa *ON DUPLICATE KEY UPDATE*, la opción *DELAYED* se ignora.

Se puede encontrar el valor usado para una columna *AUTO\_INCREMENT* usando la función [LAST\\_INSERT\\_ID\(\)](#). Desde el API C, usar la función [mysql\\_insert\\_id](#). Sin embargo, notar que las dos

funciones no se comportan de forma idéntica en todas las circunstancias.

Si se usa una sentencia *INSERT ... VALUES* con una lista de múltiples valores o *INSERT ... SELECT*, la sentencia devuelve una cadena de información con este formato:

```
Records: 100 Duplicates: 0 Warnings: 0
```

*Records* indica el número de filas procesadas por la sentencia. (No es necesariamente el número de filas insertadas. "Duplicates" puede ser distinto de cero.) "Duplicates" indica el número de filas que no pudieron ser insertadas porque contienen algún valor para un índice único ya existente. "Warnings" indica el número de intentos de inserción de valores de columnas que han causado algún tipo de problemas. Se pueden producir "Warnings" bajo cualquiera de las siguientes condiciones:

- Inserción de *NULL* en una columna declarada como *NOT NULL*. Para sentencias **INSERT** de varias filas o en sentencias **INSERT ... SELECT**, se asigna el valor por defecto apropiado al tipo de columna. Ese valor es 0 para tipos numéricos, la cadena vacía (") para tipos cadena, y el valor "cero" para tipos de fecha y tiempo.
- Asignación de un valor fuera de rango a una columna numérica. El valor se recorta al extremo apropiado del rango.
- Asignación de un valor como '10.34 a' a un campo numérico. La parte añadida se ignora y se inserta sólo la parte numérica inicial. Si el valor no tiene sentido como un número, el valor asignado es 0.
- Inserción de un valor a una columna de cadena (*CHAR*, *VARCHAR*, *TEXT* o *BLOB*) que exceda la longitud máxima para la columna. El valor se trunca a la longitud máxima de la columna.
- Inserción de un valor dentro de una columna de fecha o tiempo que sea ilegal para el tipo de columna. Se asigna a la columna el valor apropiado de cero, según su tipo.

Si se usa el API C, la cadena de información se puede obtener mediante la función [mysql\\_info](#).

Ver también: [INSERT ... SELECT](#) e [INSERT DELAYED](#)

(4.1.1)

# INSERT ... SELECT

```
INSERT [LOW_PRIORITY] [IGNORE] [INTO] tbl_name [(column_list)]
      SELECT ...
```

Con **INSERT ... SELECT**, se pueden insertar rápidamente muchas filas en una tabla desde otra u otras tablas.

Por ejemplo:

```
INSERT INTO tbl_temp2 (fld_id)
      SELECT tbl_temp1.fld_order_id
      FROM tbl_temp1 WHERE tbl_temp1.fld_order_id > 100;
```

En una sentencia **INSERT ... SELECT** se deben cumplir las siguientes condiciones :

- En versiones anteriores a MySQL 4.0.1, **INSERT ... SELECT** opera de forma implícita en el modo *IGNORE*. A partir de MySQL 4.0.1, se debe especificar *IGNORE* explícitamente para que se ignoren registros que puedas producir violaciones de claves duplicadas.
- No se debe usar *DELAYED* con **INSERT ... SELECT**.
- Antes de MySQL 4.0.14, la tabla destino de una sentencia **INSERT** no puede aparecer en la cláusula *FROM* de la parte *SELECT* de la consulta. Esta limitación se eliminó en la versión 4.0.14.
- Las columnas *AUTO\_INCREMENT* funcionan del modo normal.
- Para asegurar que el diario binario pueda ser usado para recrear las tablas originales, MySQL no permitirá inserciones concurrentes durante un **INSERT ... SELECT**.
- Actualmente, no se puede insertar en una tabla y seleccionar desde la misma tabla en una subconsulta.

Se puede usar **REPLACE** en lugar de **INSERT** para sobrescribir filas viejas. **REPLACE** es el homólogo de **INSERT IGNORE** en el tratamiento de nuevas filas que contengan valores de clave únicas que dupliquen filas existentes: Las nuevas filas se usan para reemplazar a las antiguas más que para que sean descartadas.

(4.1.1)

# INSERT DELAYED

## INSERT DELAYED ...

La opción *DELAYED* para la sentencia [INSERT](#) es una extensión de MySQL al SQL estándar que es muy práctica si se tienen clientes que no pueden esperar a que el [INSERT](#) se complete. Este es un problema frecuente cuando se usa MySQL y además se ejecutan periódicamente sentencias [SELECT](#) y [UPDATE](#) que necesitan mucho tiempo para completarse. *DELAYED* se añadió en MySQL 3.22.15.

Cuando un cliente usa **INSERT DELAYED**, obtiene una confirmación por parte del servidor de forma inmediata, y la fila se almacena en una cola para ser insertada cuando la tabla no esté en uso por ningún otro proceso.

Otra gran ventaja de usar **INSERT DELAYED** es que las inserciones desde muchos clientes son ligadas juntas y escritas en un bloque. Esto es mucho más rápido que hacer muchas inserciones separadas.

Existen algunas limitaciones en el uso de *DELAYED*:

- **INSERT DELAYED** funciona sólo con tabls **MyISAM** y **ISAM**. Para tablas **MyISAM**, si no existen bloques libres en el interior del fichero de datos, se soportan sentencias [SELECT](#) e [INSERT](#) concurrentes. Bajo esas circunstancias, raramente será necesario el uso de **INSERT DELAYED** con **MyISAM**.
- **INSERT DELAYED** debe ser usado sólo con sentecias [INSERT](#) que especifiquen listas de valores. Esto está forzado desde MySQL 4.0.18. El servidor ignora el *DELAYED* para sentencias [INSERT DELAYED ... SELECT](#).
- El servidor también ignora el *DELAYED* para sentencias [INSERT DELAYED ... ON DUPLICATE UPDATE](#).
- Ya que la sentencia regresa inmediatamente, antes de que las filas sean insertadas, no se puede usar [LAST\\_INSERT\\_ID\(\)](#) para obtener el valor *AUTO\_INCREMENT* que la sentencia generará.
- Las filas *DELAYED* no son visibles para sentencias [SELECT](#) hasta que hayan sido insertadas.

Hay que tener en cuenta que las filas actualmente en la cola se almacenan sólo en memoria hasta que sean insertadas en la tabla. Eso significa que si se fuerza la terminación de *mysqld* (por ejemplo, con kill -9) o si *mysqld* termina de forma inesperada, cualquier fila en la cola que no haya sido escrita en disco se perderá.

A continuación se describe con detalle qué ocurre cuando se usa la opción *DELAYED* con [INSERT](#) o

**REPLACE**. En esta descripción, el "proceso" es el que recibe una sentencia **INSERT DELAYED** y el "manipulador" es el proceso que manipula todas las sentencias **INSERT DELAYED** para una tabla concreta.

- Cuando un proceso ejecuta una sentencia *DELAYED* para una tabla, se crea un proceso manipulador para procesar todas las sentencias *DELAYED* para esa tabla, si es que no existe ya tal manipulador.
- El proceso verifica si el manipulador ha adquirido ya un bloqueo *DELAYED*; si no lo ha hecho, le indica al proces manipulador que lo haga. El bloqueo *DELAYED* puede ser obtenido incluso si otro proceso tiene un bloque *READ* o *WRITE* sobre la tabla. No obstante, el manipulador esperará a que terminen todos los bloqueos *ALTER TABLE* y *FLUSH TABLES* para asegurarse de que la estructura de la tabla esté lista para usarse.
- El proceso ejecuta la sentencia **INSERT**, pero en lugar de escribir la fila en la tabla, coloca una copia de la fila final en una cola que es manejada por el proceso manipulador. Cualquier error de sintaxis se notifica al proceso y enviado al proceso cliente.
- El cliente no puede obtener el número de registros duplicados desde el servidor o el valor *AUTO\_INCREMENT* de la fila resultante, porque la sentencia **INSERT** regresa antes de que la operación de inserción haya sido completada. (Si se usa el API C, la función [mysql\\_info\(\)](#) tampoco devuelve nada con significado, por la misma razón.)
- El diario binario se actualiza por el proceso manipulador cuando la fila es insertada en la tabla. En caso de inserciones de varias filas, el diario binario se acutaliza cuando la primera fila es insertada.
- Después de cada escritura de *every delayed\_insert\_limit* filas, el manipulador comprueba si hay pendiente alguna sentencia **SELECT**. Si es así, permite su ejecución antes de continuar.
- Cuando el manipulador no tiene más filas en su cola, desbloquea la tabla. Si no se reciben nuevas sentencias **INSERT DELAYED** en el intervalo de *delayed\_insert\_timeout* segundos, el manipulador termina.
- Si hay más de *delayed\_queue\_size* filas pendientes en una cola de manipulador, el proceso que ha lanzado el **INSERT DELAYED** espera hasta que haya espacio en la cola. Esto se hace para asegurarse que el servidor *mysqld* no use toda la memoria para la cola de inserciones retrasadas.
- El proceso manipulador aparece en la lista de procesos MySQL con *delayed\_insert* en la columna 'Command'. Puede ser terminado si se ejecuta la sentencia **FLUSH TABLES** o con **KILL thread\_id**. Sin embargo, antes de salir, primero almacenará todas las filas en la tabla. Durante este tiempo no aceptará ninguna nueva sentencia **INSERT** desde otro proceso. Si se ejecuta una sentencia **INSERT DELAYED** después, se creará un nuevo proceso manipulador. Esto significa que las sentencias **INSERT DELAYED** tienen mayor prioridad que las sentencias **INSERT** normales si existe un manipulador de **INSERT DELAYED** en ejecución. Otras sentencias de actualización tendrán que esperar a que la cola de **INSERT DELAYED** se vacíe, alguien finalice el proceso manipulador (con **KILL thread\_id**), o alguien ejecute **FLUSH TABLES**.
- Las siguientes variables de estado proporcionan información sobre sentencias **INSERT DELAYED**:

Variable de estado	Significado
Delayed_insert_threads	Número de procesos manipuladores
Delayed_writes	Número de filas escritas con <b>INSERT DELAYED</b>
Not_flushed_delayed_rows	Número de filas esperando a ser escritas

- Se pueden ver estas variables mediante una sentencia [SHOW STATUS](#) o ejecutando un comando *mysqladmin extended-status*.

Hay que tener en cuenta que **INSERT DELAYED** es más lenta que un [INSERT](#) normal si la tabla no está en uso. Hay algo de trabajo adicional para el servidor al manipular un proceso separado para cada tabla para la que haya inserciones de filas retrasadas. Esto significa que sólo se debe usar **INSERT DELAYED** cuando se esté realmente seguro de que es necesario.

(4.1.1)

# JOIN

MySQL soporta las siguientes sintaxis para **JOIN** para ser usadas como la parte de referencia de tabla en sentencias **SELECT** y sentencias **DELETE** y **UPDATE** multitabla:

```
table_reference, table_reference
table_reference [INNER | CROSS] JOIN table_reference [join_condition]
table_reference STRAIGHT_JOIN table_reference
table_reference LEFT [OUTER] JOIN table_reference [join_condition]
table_reference NATURAL [LEFT [OUTER]] JOIN table_reference
{ OJ table_reference LEFT OUTER JOIN table_reference
  ON conditional_expr }
table_reference RIGHT [OUTER] JOIN table_reference [join_condition]
table_reference NATURAL [RIGHT [OUTER]] JOIN table_reference
```

Donde *table\_reference* se define como:

```
tbl_name [[AS] alias]
  [[USE INDEX (key_list)]
   | [IGNORE INDEX (key_list)]
   | [FORCE INDEX (key_list)]]
```

Y *join\_condition* se define como:

```
ON conditional_expr | USING (column_list)
```

Generalmente no será necesario especificar condiciones en la parte *ON* para restringir qué filas se quieren en el conjunto de resultados, pero es mejor especificar esas condiciones en la cláusula *WHERE*. Hay algunas excepciones a esta regla.

Hay que tener en cuenta que la sintaxis *INNER JOIN* permite una *join\_condition* sólo a partir de MySQL 3.23.17. Para *JOIN* y *CROSS JOIN* sólo se permite a partir de MySQL 4.0.11.

La sintaxis { *OJ ... LEFT OUTER JOIN ...* } mostrada anteriormente sólo existe por compatibilidad con ODBC.

- Se puede definir un alias para una referencia de tabla usando *tbl\_name AS alias\_name* o

*tbl\_name alias\_name:*

```
mysql> SELECT t1.name, t2.salary FROM employee AS t1, info AS t2
->         WHERE t1.name = t2.name;
mysql> SELECT t1.name, t2.salary FROM employee t1, info t2
->         WHERE t1.name = t2.name;
```

- El condicional *ON* es cualquier expresión condicional de la forma que puede usarse en una cláusula *WHERE*.
- Si no hay ninguna fila coincidente en la tabla derecha en la parte *ON* o *USING* en un *LEFT JOIN*, se usa una fila con un valor *NULL* para todas las columnas para la tabla derecha. Se puede usar este comportamiento para encontrar registros en una tabla que no tengan contraparte en otra tabla:

```
mysql> SELECT table1.* FROM table1
->         LEFT JOIN table2 ON table1.id=table2.id
->         WHERE table2.id IS NULL;
```

Este ejemplo encuentra todas las filas en *table1* con un valor de *id* que no esté presente en *table2* (esto es, todas las filas en *table1* sin correspondencia en *table2*). Se asume que *table2.id* está declarado como *NOT NULL*.

- La cláusula *USING (column\_list)* nombra una lista de columnas que deben existir en ambas tablas. Las dos cláusulas siguientes son idénticas semánticamente:

```
a LEFT JOIN b USING (c1,c2,c3)
a LEFT JOIN b ON a.c1=b.c1 AND a.c2=b.c2 AND a.c3=b.c3
```

- La reunión *NATURAL [LEFT] JOIN* de dos tablas se define para que sea equivalente semánticamente a un *INNER JOIN* o a un *LEFT JOIN* con una cláusula *USING* que nombre todas las columnas que existan en ambas tablas.
- *INNER JOIN* y , (coma) son equivalentes semánticamente en ausencia de una condición de reunión: ambas producirán un producto cartesiano entre las tablas especificadas (esto es, todas y cada una de las filas en la primera tabla se reunirán con todas las de la segunda tabla).
- *RIGHT JOIN* trabaja de forma análoga a *LEFT JOIN*. Para mantener el código portable a través de las bases de datos, se recomienda usar *LEFT JOIN* en lugar de *RIGHT JOIN*.
- *STRAIGHT\_JOIN* es idéntico a *JOIN*, excepto que la tabla izquierda siempre es leída antes que la tabla derecha. Esto se puede usar para aquellos (pocos) casos para los cuales el optimizador de join coloca las tablas en el orden equivocado.

Desde MySQL 3.23.12, se pueden obtener pistas sobre cuáles son los índices que debe usar MySQL

cuando recupere información desde una tabla. Mediante la especificación de *USE INDEX (key\_list)*, se puede indicar a MySQL que use sólo uno de los índices posibles para encontrar filas en una tabla. La sintaxis alternativa *IGNORE INDEX (key\_list)* se puede usar para indicar a MySQL que no use algún índice particular. Estas pistas son aconsejable si [EXPLAIN](#) muestra que MySQL está usando el índice equivocado de una lista de posibles índices.

A partir de MySQL 4.0.9, se puede usar *FORCE INDEX*. Esto funciona como *USE INDEX (key\_list)* pero con el añadido de que se asumirá que un recorrido secuencial de la tabla será demasiado costoso. En otras palabras, se usará un un recorrido secuencial sólo si no hay modo de usar uno de los índices dados para encontrar filas en la tabla.

*USE KEY*, *IGNORE KEY* y *FORCE KEY* son sinónimos de *USE INDEX*, *IGNORE INDEX* y *FORCE INDEX*.

**Nota:** *USE INDEX*, *IGNORE INDEX* y *FORCE INDEX* sólo afectan a aquellos índices usados cuando MySQL decide el modo de encontrar filas en la tabla y cómo hacer la reunión. No afectan a si un índice será usado cuando se resuelva una cláusula *ORDER BY* o *GROUP BY*.

Algunos ejemplos de reunión:

```
mysql> SELECT * FROM table1,table2 WHERE table1.id=table2.id;
mysql> SELECT * FROM table1 LEFT JOIN table2 ON table1.id=table2.id;
mysql> SELECT * FROM table1 LEFT JOIN table2 USING (id);
mysql> SELECT * FROM table1 LEFT JOIN table2 ON table1.id=table2.id
-> LEFT JOIN table3 ON table2.id=table3.id;
mysql> SELECT * FROM table1 USE INDEX (key1,key2)
-> WHERE key1=1 AND key2=2 AND key3=3;
mysql> SELECT * FROM table1 IGNORE INDEX (key3)
-> WHERE key1=1 AND key2=2 AND key3=3;
```

(4.1.1)

# KILL

```
KILL thread_id
```

Cada conexión a **mysqld** se ejecuta en un hilo separado. Se puede ver qué hilo se está ejecutando con el comando **SHOW PROCESSLIST** y matarlo un hilo con el comando **KILL thread\_id**.

Si se posee el privilegio *PROCESS*, se pueden ver todos los hilos. Si se posee el privilegio *SUPER*, se pueden matar todos los hilos. En otro caso, sólo es posible ver y matar los hilos propios.

También se pueden usar los comandos *mysqladmin processlist* y *mysqladmin kill* par examinar y matar hilos.

Nota: actualmente no se puede usar el comando **KILL** con la librería del Servidor MySQL embebido, porque el servidor embebido sencillamente se ejecuta dentro de los hilos en el ordenador de la aplicación, no crea hilos de conexiones por si mismo.

Cuando se hace un **KILL**, se activa un banderín de 'kill' específico para el hilo.

En la mayoría de los casos, matar el hilo puede tomar cierto tiempo ya que el flag 'kill' se verifica sólo cada cierto tiempo.

- El bucles *ORDER BY* y *GROUP BY* del comando **SELECT**, el banderín se comprueba después de leer un bloque de filas. Si el banderín de 'kill' está activot, la sentencia se aborta.
- Cuando se realiza un **ALTER TABLE**, el banderín 'kill' se comprueba antes de que cada bloque de filas se lea desde la tabla original. Si está activo el comando se aborta y la tabla temporal se borra.
- Cuando se ejecutan sentencias **UPDATE** o **DELETE**, el banderín se comprueba después de cada lectura de bloque y después de cada actualización o borrado de fila. Si está activo, la sentencia se aborta. Si no se están usando transacciones, los cambios no podrán ser rebobinados.
- **GET\_LOCK()** abortará con *NULL*.
- Un hilo **INSERT DELAYED** almacenará rápidamente todas las filas que tenga en memoria y morirá.
- Si el hilo está en la table de manipuladores de bloqueo (estado: Locked), el bloqueo de la tabla será rápidamente abortado.
- Si el hilo está esperando a que haya espacio libre en disco en una llamada de escritura, la escritura se aborta con un mensaje de error de disco lleno.

KILL

(4.0)

# LOAD DATA

```
LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE 'file_name.txt'
[REPLACE | IGNORE]
INTO TABLE tbl_name
[FIELDS
    [TERMINATED BY '\t']
    [[OPTIONALLY] ENCLOSED BY '']
    [ESCAPED BY '\\\']
]
[LINES
    [STARTING BY '']
    [TERMINATED BY '\n']
]
[IGNORE number LINES]
[(col_name,...)]
```

La sentencia **LOAD DATA INFILE** lee filas desde un fichero de texto a una tabla a gran velocidad. Si se especifica la palabra *LOCAL*, se interpreta con respecto al cliente final de la conexión. Cuando se especifica *LOCAL*, el fichero es leído por el programa del cliente en el ordenador cliente y se envía al servidor. Si no se especifica *LOCAL*, el fichero debe estar en el ordenador servidor y es leído directamente por el servidor. (*LOCAL* está disponible desde la versión 3.22.6 de MySQL.)

Por razones de seguridad, cuando se leen ficheros de texto del servidor, los ficheros deben residir en el directorio de la base de datos o tener acceso para todos. Además, para usar **LOAD DATA INFILE** en ficheros del servidor, se debe poseer el privilegio *FILE* en el ordenador servidor.

Desde MySQL 3.23.49 y MySQL 4.0.2 (4.0.13 en Windows), *LOCAL* funcionará sólo si el servidor y el cliente han sido configurados para admitirlo. Por ejemplo, si **mysqld** se ha arrancado con *--local-infile=0*, *LOCAL* no funcionará.

Si se especifica *LOW\_PRIORITY*, la ejecución de la sentencia **LOAD DATA** se retrasará hasta que no haya otros clientes leyendo de la tabla.

Si se especifica *CONCURRENT* con una tabla **MyISAM**, entonces otros hilos pueden recuperar datos desde la tabla mientras **LOAD DATA** se está ejecutando. Usar esta opción puede afectar un poco, obviamente, al rendimiento de **LOAD DATA** salvo si no hay otros hilos usando la tabla al mismo tiempo.

Usar *LOCAL* puede ser un poco más lento que hacer la lectura desde ficheros del servidor directamente, ya que el contenido del fichero debe ser enviado a través de la conexión entre el cliente y el servidor. Por

otra parte, no se necesita el privilegio *FILE* para cargar ficheros locales.

Si se usa una versión de MySQL anterior a 3.23.24 no será posible leer desde un FIFO con **LOAD DATA INFILE**. Si se necesita leer desde un FIFO (por ejemplo la salida de gunzip), se debe usar **LOAD DATA LOCAL INFILE** en su lugar.

También se pueden cargar ficheros de datos usando la utilidad **mysqlimport**; funciona mediante el envío de un comando **LOAD DATA INFILE** al servidor. La opción *--local* hace que **mysqlimport** lea ficheros de datos desde el ordenador cliente. Se puede especificar la opción *--compress* para un mayor rendimiento sobre redes lentas si el cliente y el servidor soportan protocolos comprimidos.

Cuando se usan ficheros en el ordenador servidor, éste usa las siguientes reglas:

- Si se proporciona un camino completo, el servidor usa el camino tal cual.
- Si se usa un camino relativo con uno o más componentes previos, el servidor busca el fichero relativa al directorio de datos del servidor.
- Si se proporciona un nombre de fichero sin componentes previos, el servidor busca en el directorio de la base de datos actual.

Estas reglas implican que un fichero con el nombre './myfile.txt' se leerá desde el directorio de datos del servidor, mientras que el mismo nombre de fichero, escrito como 'myfile.txt' se leerá desde el directorio de base de datos de la base de datos actual. Por ejemplo, las siguientes sentencias **LOAD DATA** leen el fichero 'data.txt' desde el directorio de base de datos para db1 porque db1 es la base de datos actual, aunque la sentencia carga el fichero explícitamente en una tabla de la base de datos db2:

```
mysql> USE db1;
mysql> LOAD DATA INFILE "data.txt" INTO TABLE db2.my_table;
```

Las palabras clave *REPLACE* y *IGNORE* controlan la manipulación de los registros de entrada que coincidan con registros existentes con valores de clave únicos.

Si se especifica *REPLACE*, las filas de entrada reemplazan a las filas existentes (en otras palabras, filas que tengan el mismo valor para un índice primario o único que una fila existente).

Si se especifica *IGNORE*, las filas de entrada que dupliquen una fila existente en un valor de clave único serán ignoradas. Si no se especifica ninguna de las dos opciones, el comportamiento depende de si se ha especificado o no la palabra clave *LOCAL*. Sin *LOCAL*, ocurrirá un error cuando se encuentre una valor de clave duplicado, y el resto del fichero de texto será ignorado. Con *LOCAL*, el comportamiento por defecto es el mismo que si se especifica *IGNORE*; esto es porque el servidor no tiene forma de parar la transmisión del fichero en el transcurso de la operación.

Si se quiere ignorar la restricción de claves foráneas durante la carga se puede hacer **SET FOREIGN\_KEY\_CHECKS=0** antes de ejecutar **LOAD DATA**.

Si se usa **LOAD DATA INFILE** en una tabla **MyISAM** vacía, todos los índices no únicos será creados en un proceso separado (como en **REPAIR TABLE**). Normalmente, esto hace que **LOAD DATA INFILE** sea mucho más rápido cuando se tienen muchos índices. Generalmente esto es muy rápido, pero en algunos casos extremos se pueden crear los índices más rápidamente desconectándolos con **ALTER TABLE .. DISABLE KEYS** y usando **ALTER TABLE .. ENABLE KEYS** para reconstruir los índices.

**LOAD DATA INFILE** es el complemento de **SELECT ... INTO OUTFILE**. Para escribir datos desde una tabla a un fichero, usar **SELECT ... INTO OUTFILE**. Para leer el fichero de nuevo a una tabla, usar **LOAD DATA INFILE**. La sintaxis de las cláusulas *FIELDS* y *LINES* es la misma en ambos comandos. Las dos cláusulas son opcionales, pero *FIELDS* debe preceder a *LINES* si ambas se especifican.

Si se especifica la cláusula *FIELDS*, cada una de sus subcláusulas (**TERMINATED BY**, **[OPTIONALLY] ENCLOSED BY** y **ESCAPED BY**) son también opcionales, excepto que se debe especificar al menos una de ellas.

Si no se especifica una cláusula *FIELDS*, por defecto es lo mismo que si se escribe:

```
FIELDS TERMINATED BY '\t' ENCLOSED BY '' ESCAPED BY '\\'
```

Si no se especifica una cláusula *LINES*, por defecto es lo mismo que si se escribe:

```
LINES TERMINATED BY '\n'
```

Nota: si se ha generado el fichero de texto en un sistema Windows se debe cambiar lo anterior por: ***LINES TERMINATED BY '\r\n'*** ya que Windows usa dos caracteres como terminador de línea. Algunos programas, como **wordpad**, pueden usar **\r** como terminado de línea.

Si todas las líneas que se quieren leer tienen un prefijo en común que se quiere ignorar, se puede usar ***LINES STARTING BY prefix\_string*** para ello.

Es decir, por defecto **LOAD DATA INFILE** actúa como sigue:

- Busca los límites de línea en los cambios de línea.
- Si se usa ***LINES STARTING BY prefijo***, lee hasta que se encuentra el prefijo y empieza a leer en

el carácter siguiente al prefijo. Si la línea no incluye el prefijo será ignorada.

- Rompe las líneas en campos usando los tabuladores.
- No espera que los campos estén encerrados entre ningún tipo de comillas.
- Interpreta los tabuladores, retornos de línea o '\' precedidos por '\' como caracteres literales que forman parte de los valores de campos.

Por el contrario, por defecto [SELECT ... INTO OUTFILE](#):

- Escribe tabuladores entre campos.
- No encierra los campos entre comillas.
- Usa '\' para escapar las apariciones de tabuladores, cambios de línea o '\' que ocurran dentro de valores de campos.
- Escribe cambios de línea al final de las líneas.

Para escribir *FIELDS ESCAPED BY '\\'*, se deben especificar dos barras para el valor que se leerá como una barra sencilla.

La opción *IGNORE número LINES* se puede usar para ignorar líneas al comienzo del fichero. Por ejemplo, se puede usar *IGNORE 1 LINES* para ignorar una línea inicial de cabecera que contenga los nombres de columnas:

```
mysql> LOAD DATA INFILE "/tmp/file_name" INTO TABLE test IGNORE 1 LINES;
```

Cuando se usa [SELECT ... INTO OUTFILE](#) en un tande con **LOAD DATA INFILE** para escribir datos desde una base de datos a un fichero y a continuación leer el fichero de nuevo en la base de datos, las opciones de manipulación de campos y líneas para ambos comandos deben coincidir. De otro modo, **LOAD DATA INFILE** puede no interpretar el contenido del fichero apropiadamente. Supongamos que se usa [SELECT ... INTO OUTFILE](#) para escribir un fichero con campos delimitados con comas:

```
mysql> SELECT * INTO OUTFILE 'data.txt'
->      FIELDS TERMINATED BY ','
->      FROM ...;
```

Para leer el fichero delimitado con comas, la sentencia correcta será:

```
mysql> LOAD DATA INFILE 'data.txt' INTO TABLE table2
->      FIELDS TERMINATED BY ',';
```

Si en lugar de eso se intenta leer el fichero con la sentencia mostrada abajo, no funcionará porque indica a **LOAD DATA INFILE** que lea usando tabuladores entre los campos:

```
mysql> LOAD DATA INFILE 'data.txt' INTO TABLE table2
->          FIELDS TERMINATED BY '\t';
```

El resultado probable es que cada línea leída se interprete como un único campo.

**LOAD DATA INFILE** puede usarse también para leer ficheros obtenidos desde fuentes externas. Por ejemplo, un fichero en formato **dBASE** puede tener campos separados con comas y encerrados entre comillas dobles. Si las líneas del fichero terminan con caracteres de cambio de línea, el comando siguiente ilustra las opciones de manipulación de campos y líneas que se deben usar para cargar el fichero:

```
LOAD DATA INFILE 'data.txt' INTO TABLE tbl_name
->          FIELDS TERMINATED BY ',' ENCLOSED BY '"'
->          LINES TERMINATED BY '\n';
```

Cualquiera de las opciones de manipulación, de campo o de línea puede ser una cadena vacía ("). Si no está vacía, los valores para *FIELDS [OPTIONALLY] ENCLOSED BY* y *FIELDS ESCAPED BY* deben ser un único carácter. Los valores *FIELDS TERMINATED BY* y *LINES TERMINATED BY* pueden ser más de un carácter. Por ejemplo, para escribir líneas que terminen con parejas de retorno de línea y avance de línea o para leer un fichero que contenga esas líneas, se especifica una cláusula *LINES TERMINATED BY '\r\n'*.

Por ejemplo, para leer un fichero con bromas, que están separadas con una línea de %, dentro de una tabla SQL se puede hacer:

```
CREATE TABLE jokes (a INT NOT NULL AUTO_INCREMENT PRIMARY KEY, joke
TEXT NOT NULL);
LOAD DATA INFILE "/tmp/jokes.txt" INTO TABLE jokes FIELDS TERMINATED BY ""
LINES TERMINATED BY "\n%%\n" (joke);
```

*FIELDS [OPTIONALLY] ENCLOSED BY* controla el entrecomillado de campos. Para salida (**SELECT ... INTO OUTFILE**), si se omite la palabra *OPTIONALLY*, todos los campos se encerrarán entre el carácter *ENCLOSED BY*. Un ejemplo de esa salida así (usando una coma como delimitador de campo) es este:

```
"1","a string","100.20"
"2","a string containing a , comma","102.20"
"3","a string containing a \" quote","102.20"
"4","a string containing a \", quote and comma","102.20"
```

Si se especifica *OPTIONALLY*, el carácter *ENCLOSED BY* se usa sólo para encerrar campos de tipos *CHAR* y *VARCHAR*:

```
1,"a string",100.20
2,"a string containing a , comma",102.20
3,"a string containing a \" quote",102.20
4,"a string containing a \", quote and comma",102.20
```

Se puede ver que las ocurrencias del carácter *ENCLOSED BY* dentro del valor de campo se escapan precediéndolas con el carácter *ESCAPED BY*. También se puede ver que si se especifica un valor vacío en *ESCAPED BY*, es posible generar una salida que no puede ser leída apropiadamente por **LOAD DATA INFILE**. Por ejemplo, la salida anterior se muestra a continuación tal como aparecería si el carácter de escape fuese vacío. El segundo campo en la cuarta línea contiene una coma seguida de unas comillas dobles, el cual (erroneamente) aparenta la terminación de un campo:

```
1,"a string",100.20
2,"a string containing a , comma",102.20
3,"a string containing a " quote",102.20
4,"a string containing a ", quote and comma",102.20
```

Para la entrada, el carácter *ENCLOSED BY*, si está presente, se extrae de los extremos de los valores de campo. (Esto es cierto siempre que se especifique *OPTIONALLY*; *OPTIONALLY* no tiene efecto en la interpretación de entradas.) Las apariciones del carácter *ENCLOSED BY* precedidas por el carácter *ESCAPED BY* se interpretan como parte del valor del campo actual.

Si el campo empieza con el carácter *ENCLOSED BY*, las apariciones de ese carácter se reconocen como el valor de terminación del campo sólo si va seguido por el separador de campo o por la secuencia de *TERMINATED BY*. Para impedir ambigüedad, las apariciones del carácter *ENCLOSED BY* en el interior de un valor de campo pueden ser duplicadas y serán interpretadas como una aparición sencilla del carácter. Por ejemplo, si se especifica *ENCLOSED BY ''*, las comillas se manipulan así:

```
"The ""BIG"" boss"  -> The "BIG" boss
The "BIG" boss      -> The "BIG" boss
The ""BIG"" boss    -> The ""BIG"" boss
```

*FIELDS ESCAPED BY* controla cómo escribir o leer caracteres especiales. Si el carácter *FIELDS ESCAPED BY* no es vacío, se usa como prefijo para los siguientes caracteres en las salidas:

- El carácter *FIELDS ESCAPED BY*.
- El carácter *FIELDS [OPTIONALLY] ENCLOSED BY*.
- El primer carácter de los valores *FIELDS TERMINATED BY* y *LINES TERMINATED BY*.
- ASCII 0 (que se escribe actualmente siguiendo el carácter de escape por ASCII '0', y no un byte de valor cero).

Si el carácter *FIELDS ESCAPED BY* es vacío, no se escapan caracteres. Probablemente no sea una buena idea especificar un carácter vacío como carácter de escape, particularmente si los valores de campos en los datos contienen cualquiera de los caracteres en la lista anterior.

Para entrada, si el carácter *FIELDS ESCAPED BY* no es vacío, las apariciones de ese carácter se eliminan y se toma el siguiente carácter literalmente como parte del valor del campo. Las excepciones son los caracteres '0' y 'N' (por ejemplo, \0 o \N si el carácter de escape es '\'). Estas secuencias se interpretan como ASCII 0 (un byte de valor cero) y NULL. Mirar más abajo para ver las reglas de manipulación de NULL.

En ciertos casos, las opciones de manipulación de campos y líneas interactúan:

- Si *LINES TERMINATED BY* es una cadena vacía y *FIELDS TERMINATED BY* es un carácter no vacío, las líneas son terminadas también con el carácter *FIELDS TERMINATED BY*.
- Si los valores para *FIELDS TERMINATED BY* y *FIELDS ENCLOSED BY* son ambos vacíos ("), se usa un formato de fila fija (no delimitada). Con el formato de fila fija, no se usan delimitadores entre campos (pero aún es posible tener un terminador de línea). En su lugar, los valores de columna se escriben y leen usando las anchuras de visualización de las columnas. Por ejemplo, si una columna se declara como INT(7), los valores para la columna se escriben usando campos de siete caracteres. En la entrada, los valores para la columna se obtienen leyendo siete caracteres. *LINES TERMINATED BY* se sigue usando para separar líneas. Si una línea no contiene todos los campos, el resto serán asignados a sus valores por defecto. Si no se dispone de un terminador de línea, se debe asignar a ". En ese caso, el fichero de texto debe contener todos los campos para cada fila. El formato de fila fija también afecta a la manipulación de valores NULL; ver abajo. El formato de fila fija no funcionará si se usa un conjunto de caracteres multibyte.

La manipulación de valores NULL varía dependiendo de las opciones para *FIELDS* y *LINES* que se usen:

- Para los valores por defecto de *FIELDS* y *LINES*, NULL se escribe como \N para salida y \N se lee como NULL en entrada (asumiendo que el carácter *ESCAPED BY* es '\').
- Si *FIELDS ENCLOSED BY* no es vacío, un campo que contenga la palabra NULL como valor

(esto es diferente si la palabra NULL está encerrada entre caracteres *FIELDS ENCLOSED BY*, el cual se lee como la cadena 'NULL').

- Si *FIELDS ESCAPED BY* es vacío, NULL se escribe como la palabra NULL.
- Con el formato de fila fija (que es cuando *FIELDS TERMINATED BY* y *FIELDS ENCLOSED BY* son ambos vacíos), NULL se escribe como una cadena vacía. Esto hace que tanto los valores NULL como las cadenas vacías no se puedan diferenciar cuando se escriben en el fichero, ya que ambos se escriben como cadenas vacías. Si se necesita diferenciar cuando se lee entre ambos casos, no se debe usar el formato de fila fija.

Algunos casos no son soportados por **LOAD DATA INFILE**:

- Filas de tamaño constante (*FIELDS TERMINATED BY* y *FIELDS ENCLOSED BY* vacíos) y columnas de tipo *BLOB* o *TEXT*.
- Si se especifica un separador que es el mismo o un prefijo de otro, **LOAD DATA INFILE** no podrá interpretar la entrada de forma adecuada. Por ejemplo, las siguientes cláusulas *FIELDS* causarán problemas:

```
FIELDS TERMINATED BY '' ENCLOSED BY ''
```

- Si *FIELDS ESCAPED BY* es vacía, un valor de campo que contenga el carácter *FIELDS ENCLOSED BY* o *LINES TERMINATED BY* seguido del valor de *FIELDS TERMINATED BY* provocará que **LOAD DATA INFILE** dejen de leer un campo o línea demasiado pronto. Esto sucede porque **LOAD DATA INFILE** no puede determinar dónde termina el valor de campo o de línea.

El ejemplo siguiente carga todas las columnas de la tabla persondata:

```
mysql> LOAD DATA INFILE 'persondata.txt' INTO TABLE persondata;
```

No se especifica ninguna lista de campos, de modo que **LOAD DATA INFILE** espera que las filas de entrada contengan un campo para cada columna de la tabla. Se usan los valores por defecto para *FIELDS* y *LINES*.

Si se desea cargar sólo algunas columnas de la tabla, se debe especificar una lista de campos:

```
mysql> LOAD DATA INFILE 'persondata.txt'
-> INTO TABLE persondata (col1,col2,...);
```

Se debe especificar una lista de campos si el orden de los campos dentro del fichero de entrada es distinto del orden de las columnas en la tabla. De otro modo, MySQL no puede hacer coincidir cada campo de entrada con su columna en la tabla.

Si una fila tiene pocos campos, las columnas para las que no exista campo de entrada tomarán sus valores por defecto.

Un valor de campo vacío se interpreta de forma diferente que si el valor del campo se ha perdido:

- Para tipos cadena, se asigna una cadena vacía a la columna.
- Para tipos numéricos, se asigna 0 a la columna.
- Para tipos de fecha y tiempo, se asigna el valor de "cero" apropiado según el tipo.

Estos son los mismos valores que se asignan si se asigna una cadena vacía explícitamente a una cadena, un número o a un tipo fecha o hora en una sentencia [INSERT](#) o [UPDATE](#).

A las columnas *TIMESTAMP* sólo se les asigna la fecha y hora actual si hay un valor NULL para la columna (es decir, \N), o (sólo para la primera columna *TIMESTAMP*) si la columna *TIMESTAMP* se omite de la lista de campos cuando se especifica una lista de campos.

Si una fila de entrada tiene demasiados campos, los campos extra serán ignorados y el número de avisos se incrementa. Antes de la versión 4.1.1 de MySQL, los avisos son sólo un número que indican que algo fue mal. En MySQL 4.1.1 se puede usar [SHOW WARNINGS](#) para obtener más información sobre qué ha ido mal.

**LOAD DATA INFILE** tiene preferencia por las entradas como cadenas, de modo que no se pueden usar valores numéricos para columnas *ENUM* o *SET*, modo que sí es aplicable a sentencias [INSERT](#). Todos los valores *ENUM* y *SET* deben ser especificados como cadenas.

Si se usa el API C, se puede obtener información sobre la consulta llamando a la función del API [mysql\\_info](#) cuando la consulta **LOAD DATA INFILE** finaliza. El formato de la cadena de información es:

```
Records: 1 Deleted: 0 Skipped: 0 Warnings: 0
```

Los avisos (warnings) ocurren en las mismas circunstancias que cuando los valores son insertados vía una sentencia [INSERT](#), excepto que **LOAD DATA INFILE** también genera avisos cuando hay pocos o demasiados campos en una fila de entrada. Los avisos no son almacenados; el número de avisos sólo pueden ser usados como una indicación de que todo fue bien.

Si se obtienen avisos y se quiere saber exactamente por qué, una forma de hacerlo es usar [SELECT ... INTO OUTFILE](#) en otro fichero y compararlo con el original.

Si se necesita usar **LOAD DATA** para leer desde un pipe, se puede usar el siguiente truco:

```
mkfifo /mysql/db/x/x
chmod 666 /mysql/db/x/x
cat < /dev/tcp/10.1.1.12/4711 > /nt/mysql/db/x/x
mysql -e "LOAD DATA INFILE 'x' INTO TABLE x" x
```

Si se usa la versión 3.23.25 de MySQL o anterior sólo se puede hacer lo anterior con **LOAD DATA LOCAL INFILE**.

En MySQL 4.1.1 se puede usar [SHOW WARNINGS](#) para obtener una lista de los primeros max\_error\_count avisos.

(4.0)

# LOCK TABLES

## UNLOCK TABLES

```
LOCK TABLES tbl_name [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}
            [, tbl_name [AS alias] {READ [LOCAL] | [LOW_PRIORITY]
WRITE} ...]
...
UNLOCK TABLES
```

**LOCK TABLES** bloquea tablas para el hilo actual. **UNLOCK TABLES** libera cualquier bloqueo para el hilo actual. Todas las tablas bloqueadas por el hilo actual serán implícitamente desbloqueadas cuando el hilo realice otro **LOCK TABLES**, o cuando la conexión con el servidor se cierre.

Para usar **LOCK TABLES** en MySQL 4.0.2 se necesita el privilegio global **LOCK TABLES** y un privilegio *SELECT* en las tablas involucradas. En MySQL 3.23 se necesitan los privilegios *SELECT*, *INSERT*, *DELETE* y *UPDATE* para las tablas.

El principal motivo para usar **LOCK TABLES** es para la emulación de transacciones o para obtener mayor velocidad en la actualización de tablas. Esto se explica con más detalle más tarde.

Si el hilo obtiene un bloqueo *READ* en la tabla, ese hilo (y todos los demás) sólo pueden leer de la tabla. Si obtiene un bloqueo *WRITE*, entonces sólo el hilo que tiene el bloqueo puede leer de o escribir en la tabla. El resto de los hilos serán bloqueados.

La diferencia entre *READ LOCAL* y *READ* es que *READ LOCAL* permite ejecutar sentencias [INSERT](#) no conflictivas mientras el bloqueo esté activo. Esto no podrá ser usado si se van a manipular los ficheros de la base de datos fuera de MySQL mientras se mantiene el bloqueo.

Cuando se usa **LOCK TABLES**, se deben bloquear todas las tablas que se vayan a usar y se debe usar el mismo alias que se vaya a usar en las consultas. Si se está usando una tabla muchas veces en una consulta (con alias), se debe tener un bloqueo 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:

1. Ordena todas las tablas a bloquear en un orden interno definido (desde el punto de vista del usuario el orden es indefinido).
2. Si una tabla está bloqueada para lectura y escritura, pone el bloqueo de escritura antes que el de lectura.
3. 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 libere. 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 [INSERT DELAYED](#), porque en ese caso [INSERT](#) se ejecuta en un hilo separado.

Normalmente, no se deben bloquear tablas, ya que todas las sentencias [UPDATE](#) simples son atómicas; ningún otro hilo puede interferir con otro que esté ejecutando actualmente una sentencia 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 escribe 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 [SELECT](#) y una [UPDATE](#). El siguiente ejemplo requiere **LOCK TABLES** para que se ejecute de forma segura:

```
mysql> LOCK TABLES trans READ, customer WRITE;
mysql> SELECT SUM(value) FROM trans WHERE customer_id=some_id;
mysql> UPDATE customer SET total_value=sum_from_previous_statement
    ->      WHERE customer_id=some_id;
mysql> UNLOCK TABLES;
```

Sin **LOCK TABLES**, hay una oportunidad de que otro hilo pueda insertar una nueva fila en la tabla "trans" entre la ejecución de las sentencias SELECT y UPDATE.

Usando actualizaciones graduales (UPDATE *customer SET value=value+new\_value*) o la función LAST\_INSERT\_ID, se puede evitar el uso de **LOCK TABLES** en muchos casos.

También se pueden resolver algunos casos 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 u 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 FLUSH 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.

(4.0)

# OPTIMIZE TABLE

```
OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name[,tbl_name]...
```

**OPTIMIZE TABLE** debe usarse si se ha eliminado gran parte de una tabla o si se han hecho muchos cambios en una tabla con filas de tamaño variable (tablas que contengan columnas *VARCHAR*, *BLOB* o *TEXT*). Los registros borrados se mantienen en una lista enlazada y subsiguientes operaciones [INSERT](#) hacen uso de las posiciones de anteriores registros. Se puede usar **OPTIMIZE TABLE** to recuperar el espacio no usado y para desfragmentar el fichero de datos.

En la mayoría de los sistemas no será necesario ejecutar **OPTIMIZE TABLE**. Aunque se hagan muchas actualizaciones en filas de longitud variable no es probable que sea necesario hacerlo más de una vez al mes o a la semana, y sólo en ciertas tablas.

De momento, **OPTIMIZE TABLE** funciona sólo en tablas **MyISAM** y **BDB**. Para tablas **BDB**, **OPTIMIZE TABLE** se convierte en [ANALYZE TABLE](#).

Se puede hacer que **OPTIMIZE TABLE** funcione con otros tipos de tabla arrancando **mysqld** con `--skip-new` o `--safe-mode`, pero en ese caso **OPTIMIZE TABLE** se ejecuta como [ALTER TABLE](#).

**OPTIMIZE TABLE** trabaja del modo siguiente:

- Si la tabla contiene filas borradas o divididas, repara la tabla.
- Si las páginas de índices no están ordenadas, las ordena.
- Si las estadísticas no están al día (y la reparación no puede realizarse ordenando el índice), las actualiza.

La tabla se bloquea durante el tiempo en que se ejecuta **OPTIMIZE TABLE**.

Antes de MySQL 4.1.1, los comandos **OPTIMIZE** no actualizaban el diario binario. Desde MySQL 4.1.1 lo hacen salvo que se use la palabra clave opcional *NO\_WRITE\_TO\_BINLOG* (o si alias *LOCAL*).

(4.0)

# RENAME TABLE

```
RENAME TABLE tbl_name TO new_tbl_name  
[, tbl_name2 TO new_tbl_name2] ...
```

Esta sentencia renombra una o más tablas. Se añadió en MySQL 3.23.23.

La operación de renombrado se hace atómicamente, lo que significa que ningún otro proceso puede acceder a ninguna de las tablas mientras se realiza el renombrado. Por ejemplo, si se tiene una tabla existente `old_table`, se puede crear otra tabla `new_table` que tenga la misma estructura pero que esté vacía, y entonces reemplazar la tabla existente con la vacía tal como sigue:

```
CREATE TABLE new_table (...);  
RENAME TABLE old_table TO backup_table, new_table TO old_table;
```

Si la sentencia renombra más de una tabla, las operaciones se realizan de izquierda a derecha. Si se quiere intercambiar el nombre de dos tablas, se puede hacer de este modo (asumiendo que no existe ninguna tabla con el nombre `tmp_table`):

```
RENAME TABLE old_table TO tmp_table,  
             new_table TO old_table,  
             tmp_table TO new_table;
```

Mientras dos bases de datos estén en el mismo sistema de ficheros, también es posible renombrar una tabla para moverla desde una base de datos a otra:

```
RENAME TABLE current_db.tbl_name TO other_db.tbl_name;
```

Cuando se ejecuta **RENAME**, no se puede tener ninguna tabla bloqueada o transacciones activas. Además, se deben tener los privilegios *ALTER* y *DROP* en la tabla original, y los privilegios *CREATE* e *INSERT* en la nueva.

Si MySQL encuentra cualquier error en un renombrado múltiple, hará un renombrado inverso para todas las tablas renombradas para dejar las tablas con los nombres originales.

(4.1.1)

# REPAIR TABLE

```
REPAIR [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name[,tbl_name...]
[QUICK] [EXTENDED] [USE_FRM]
```

**REPAIR TABLE** funciona sólo con tablas **MyISAM** y es lo mismo que ejecutar *myisamchk -r table\_name* en la tabla.

Normalmente nunca se tendrá que ejecutar este comando, pero si ocurre un desastre es muy probable que se puedan recuperar todos los datos de una tabla **MyISAM** con **REPAIR TABLE**. Si las tablas están muy corruptas, se debe intentar encontrar el motivo, para eliminar la necesidad de usar **REPAIR TABLE**.

**REPAIR TABLE** repara una tabla posiblemente corrupta. El comando devuelve una tabla con las columnas siguientes:

Columna	Valor
Table	Nombre de tabla
Op	Siempre "repair" (reparar)
Msg_type	Uno de "status", "error", "info" o "warning" (estado, error, información o aviso)
Msg_text	El mensaje

La sentencia puede producir muchas filas de información para cada tabla reparada. La última fila será de estado y debe ser normalmente OK. Si no se obtiene un OK, se debe intentar reparar la tabla con *myisamchk --safe-recover*, ya que **REPAIR TABLE** aún no tiene implementadas todas las opciones de *myisamchk*. En un futuro cercano, será más flexible.

Si se usa *QUICK*, **REPAIR TABLE** intenta reparar sólo el árbol de índices.

Si se usa *EXTENDED*, MySQL creará el índice fila a fila en lugar de crear un índice de una vez mediante ordenamiento; esto puede ser mejor que ordenar en claves de longitud constante si se tienen claves *CHAR* largas que se comprimen bien. Este tipo de reparación se como si se usase *myisamchk --safe-recover*.

Desde MySQL 4.0.2, existe un modo *USE\_FRM* para **REPAIR**. Hay que usarlo si el fichero '.MYI' se ha perdido o si su cabecera está corrupta. En este modo MySQL recreará la tabla, usando información del fichero '.frm'. Este tipo de reparación no se puede hacer con *myisamchk*.

Cuidado: si **mysqld** muere durante un **REPAIR TABLE**, es esencial que lo primero que se haga sea otro **REPAIR** en la tabla antes de ejecutar cualquier otro comando en ella. (Por supuesto, es siempre mejor arrancar con un backup). En el peor caso se puede obtener un fichero de índices limpio sin información sobre el fichero de datos y con el siguiente comando se sobrescribe el fichero de datos. No es probable, pero es posible.

Antes de MySQL 4.1.1, los comandos **REPAIR** no actualizaban el diario binario. Desde MySQL 4.1.1 sí se hace, salvo que se use la palabra opcional *NO\_WRITE\_TO\_BINLOG* (o su alias *LOCAL*).

(4.0)

# REPLACE

```
REPLACE [LOW_PRIORITY | DELAYED]
  [INTO] tbl_name [(col_name,...)]
VALUES ({expr | DEFAULT},...),(...),...
```

O:

```
REPLACE [LOW_PRIORITY | DELAYED]
  [INTO] tbl_name
SET col_name={expr | DEFAULT}, ...
```

O:

```
REPLACE [LOW_PRIORITY | DELAYED]
  [INTO] tbl_name [(col_name,...)]
SELECT ...
```

**REPLACE** trabaja exactamente igual que [INSERT](#), excepto que si existe algún registro viejo en la tabla que tenga el mismo valor que uno nuevo para un índice *PRIMARY KEY* o *UNIQUE*, el viejo se borra antes de que el nuevo sea insertado.

Hay que tener en cuenta que salvo que la tabla tenga una *PRIMARY KEY* o un índice *UNIQUE*, usar una sentencia **REPLACE** no tiene sentido. En ese caso es equivalente usar una sentencia [INSERT](#), ya que no hay ningún índice que se pueda usar para determinar si una nueva fila duplica a otra.

Los valores para todas las columnas se toman de los valores especificados en la sentencia **REPLACE**. A cualquier columna perdida se le asigna su valor por defecto, justo lo mismo que ocurre con [INSERT](#). No es posible referirse a los valores de la columna vieja y usarlos en la nueva. Podría parecer que era posible hacerlo en algunas versiones anteriores de MySQL, pero se trataba de un error que ha sido corregido.

Para poder usar **REPLACE**, se deben poseer los privilegios *INSERT* y *DELETE* para la tabla.

Las sentencias **REPLACE** devuelve un contador para indicar el número de filas afectadas. Ese número es la suma de filas borradas e insertadas. Si el contador es 1 para un **REPLACE** de una única fila, la fila

fue insertada y no se borró ninguna fila. Si el contador es mayor de 1, una o más de las viejas filas fue borrada antes de que la nueva fila fuese insertada. Es posible que una única fila reemplace a más de una fila vieja si la tabla contiene varios índices únicos y la nueva fila duplica valores de diferentes filas viejas en diferentes índices únicos.

El contador de filas afectadas hace sencillo determinar si **REPLACE** sólo ha añadido una fila o si ha reemplazado alguna fila: Comprobar si el contador es 1 (añadida) o mayor (reemplazada).

Si se está usando el API C, el contador de filas afectadas se puede obtener usando la función [mysql\\_affected\\_rows\(\)](#).

Actualmente, no es posible reemplazar en una tabla y seleccionar de la misma tabla en una subconsulta.

A continuación se muestra con más detalle el algoritmo que se usa (también se usa con [LOAD DATA ... REPLACE](#)):

1. Intenta insertar la nueva fila en la tabla.
2. Mientras la inserción falle porque ocurra un error de clave duplicada para una clave primaria o única:
  1. Borrar de la tabla la fila conflictiva que que tenga el valor de clave duplicado.
  2. Intentar la inserción de la nueva fila en la tabla otra vez.

(4.1.1)

# RESET

```
RESET reset_option [,reset_option] ...
```

El comando **RESET** se usa para limpiar el estado de varias operaciones del servidor. También actúa como una versión más fuerte del comando **FLUSH**.

Para ejecutar **RESET**, se debe tener el privilegio *RELOAD*.

*reset\_option* puede tomar uno de los siguientes valores:

Opción	Descripción
MASTER	Borra todos los diarios binarios listados en el fichero de índices, reseteando el fichero de índices de diarios binarios para vaciarlo. Antes se llamaba <b>FLUSH MASTER</b> .
QUERY CACHE	Elimina todos los resultados de consulta del caché de consultas.
SLAVE	Hace que el proceso esclavo olvide su posición en los diarios binarios maestros. Antes se llamaba <b>FLUSH SLAVE</b> .

(4.1.1)

# SELECT

```

SELECT
  [ALL | DISTINCT | DISTINCTROW]
  [HIGH_PRIORITY]
  [STRAIGHT_JOIN]
  [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
  [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
select_expr,...
[INTO OUTFILE 'file_name' export_options]
 | INTO DUMPFILE 'file_name']
[FROM table_references
  [WHERE where_definition]
  [GROUP BY {col_name | expr | position}
    [ASC | DESC], ... [WITH ROLLUP]]
  [HAVING where_definition]
  [ORDER BY {col_name | expr | position}
    [ASC | DESC] ,... ]
  [LIMIT {[offset,] row_count | row_count OFFSET offset}]
  [PROCEDURE procedure_name(argument_list)]
  [FOR UPDATE | LOCK IN SHARE MODE]]

```

**SELECT** se usa para recuperar filas seleccionadas de una o más tablas. El soporte para sentencias *UNION* y subconsultas está disponible a partir de MySQL 4.0 y 4.1, respectivamente.

- Cada *select\_expr* indica una columna que se quiere recuperar.
- *table\_references* indica la tabla o tablas de las que se recuperan filas. Su sintaxis se describe en [JOIN](#).
- *where\_definition* consiste de la palabra clave *WHERE* seguida por una expresión que indica la condición o condiciones que las filas deben satisfacer para ser seleccionadas.

**SELECT** puede usarse también para recuperar filas calculadas sin referencia a ninguna tabla. Por ejemplo:

```

mysql> SELECT 1 + 1;
      -> 2

```

Todas las cláusulas usadas deben darse en el mismo orden exacto que se muestra en la descripción de la sintaxis. Por ejemplo, la cláusula *HAVING* debe estar después de cualquier cláusula *GROUP BY* y antes de cualquier cláusula *ORDER BY*.

- Una *select\_expr* puede usar alias mediante *AS nombre\_alias*. El alias se usa como un nombre de columna en expresiones y puede usarse por las cláusulas *ORDER BY* o *HAVING*. Por ejemplo:

```
mysql> SELECT CONCAT(apellido,', ', nombre) AS nombre_completo
        FROM mitabla ORDER BY nombre_completo;
```

- La palabra clave *AS* es opcional cuando se define un alias en una *select\_expr*. El ejemplo anterior se puede escribir como:

```
mysql> SELECT CONCAT(apellido,', ', nombre) nombre_completo
        FROM mitabla ORDER BY nombre_completo;
```

- Debido a que *AS* es opcional, puede ocurrir un problema si se olvida la coma entre dos *select\_expr*: MySQL interpreta el segundo como un alias. Por ejemplo, en la sentencia siguiente, *columnab* se trata como un alias:

```
mysql> SELECT columna columnab FROM mitabla;
```

- No está permitido usar un alias en una cláusula *WHERE*, porque el valor de la columna puede que no esté determinado todavía cuando la cláusula *WHERE* es ejecutada.
- La cláusula *FROM table\_references* indica las tablas desde las que se recuperarán filas. Si se nombra más de una tabla, se realiza una unión ([JOIN](#)). Para cada tabla especificada, opcionalmente se puede especificar un alias.

```
table_name [[AS] alias]
           [[USE INDEX (key_list)]
           | [IGNORE INDEX (key_list)]
           | FORCE INDEX (key_list)]]
```

El uso de *USE INDEX*, *IGNORE INDEX*, *FORCE INDEX* para proporcionar al optimizador pistas sobre cómo elegir índices se describe en la sintaxis de [JOIN](#). En MySQL 4.0.14, se puede usar [SET max\\_seeks\\_for\\_key=value](#) como una alternativa para forzar a MySQL a elegir un recorrido secuencial por clave en lugar de un recorrido secuencial de la tabla.

Desde la versión 3.23.12 de MySQL, se pueden obtener pistas sobre qué índice debe usar

- Se puede hacer referencia a una tabla con el nombre de la tabla "tbl\_name" (dentro de la base de datos actual), o con la especificación completa incluyendo la base de datos "dbname.tbl\_name".

También se puede hacer referencia a una columna como "col\_name", "tbl\_name.col\_name", o "db\_name.tbl\_name.col\_name". No es necesario especificar un prefijo "tbl\_name" o "db\_name.tbl\_name" para referenciar una columna en una sentencia **SELECT** a no ser que la referencia pueda resultar ambigua.

- Desde la versión 4.1.0, se puede especificar *DUAL* como nombre de una tabla vacía, en situaciones en las que no haya tablas definidas.

```
mysql> SELECT 1 + 1 FROM DUAL;
-> 2
```

Esta es una característica añadida sólo por compatibilidad. Ciertos servidores requieren esa sintaxis.

- Se puede definir un alias a una referencia de tabla mediante tbl\_name [AS] alias\_name:

```
mysql> SELECT t1.name, t2.salary FROM employee AS t1, info AS t2
-> WHERE t1.name = t2.name;
mysql> SELECT t1.name, t2.salary FROM employee t1, info t2
-> WHERE t1.name = t2.name;
```

- En la cláusula *WHERE*, se puede usar cualquiera de las funciones soportadas por MySQL, excepto funciones las de reunión (resumen).
- Las columnas seleccionadas pueden ser referenciadas a cláusulas *ORDER BY* y *GROUP BY* usando nombres de columna, alias de columna o posiciones de columna. Las posiciones de columna son enteros que empiezan en 1:

```
mysql> SELECT college, region, seed FROM tournament
-> ORDER BY region, seed;
mysql> SELECT college, region AS r, seed AS s FROM tournament
-> ORDER BY r, s;
mysql> SELECT college, region, seed FROM tournament
-> ORDER BY 2, 3;
```

Para ordenar en orden inverso se añade la palabra clave *DESC* (descendente) al nombre de la columna en la cláusula *ORDER BY* en la que se está ordenando. Por defecto el orden es ascendente, pero puede ser especificado explícitamente por la palabra clave *ASC*. El uso de posiciones de columna está desaconsejado ya que esa sintaxis ha sido eliminada de SQL estándar.

- Si se usa *GROUP BY*, la filas de salida serán ordenadas de acuerdo con el *GROUP BY* como si se hubiese usado *ORDER BY* sobre los campos del *GROUP BY*. MySQL ha extendido la cláusula *GROUP BY* a partir de la versión 3.23.34 de modo que se puede especificar también *ASC* y

*DESC* después de los nombres de columna en la cláusula:

```
SELECT a,COUNT(b) FROM test_table GROUP BY a DESC
```

- MySQL ha extendido el uso de *GROUP BY* para permitir seleccionar campos que no se han mencionado en la cláusula *GROUP BY*. Si no se obtiene el resultado esperado de la consulta, leer la descripción de *GROUP BY*.
- A partir de MySQL 4.1.1, *GROUP BY* permite el modificador *WITH ROLLUP*.
- La cláusula *HAVING* se aplica cerca del final, justo antes de que los resultados se envíen al cliente, sin optimizaciones. (*LIMIT* se aplica después de *HAVING*). Antes de MySQL 5.0.2, una cláusula *HAVING* se puede referir a cualquier columna o alias en la *select\_expr* de lista **SELECT** o en las subconsultas exteriores, y a las funciones agregadas. SQL estándar requiere que *HAVING* debe hacer deferencia sólo a columnas en la cláusula *GROUP BY* o columnas usadas en funciones agregadas. Para permitir ambos comportamientos, el de SQL estándar y el específico de MySQL, que permite referirse a columnas en la lista **SELECT**, a partir de MySQL 5.0.2 se permite que *HAVING* se refiera a columnas en la lista **SELECT**, columnas en la cláusula *GROUP BY*, columnas en subconsultas exteriores, y a funciones agregadas. Por ejemplo, la siguiente sentencia funciona en MySQL 5.0.2, pero produce un error en versiones anteriores:

```
mysql> SELECT COUNT(*) FROM t GROUP BY col1 HAVING col1 = 2;
```

Si la cláusula *HAVING* se refiere a una columna que es ambigua, se produce un aviso. En la sentencia siguiente, *col2* es ambiguo porque se usa tanto como un alias y como un nombre de columna:

```
mysql> SELECT COUNT(col1) AS col2 FROM t GROUP BY col2 HAVING col2 = 2;
```

Se da preferencia al comportamiento de SQL estándar, así que si un nombre de columna en un *HAVING* se usa en un *GROUP BY* y como un alias de columna en la lista de columnas de salida, se toma preferentemente la columna en *GROUP BY*.

- No se debe usar *HAVING* para items para los que se pueda usar una cláusula *WHERE*. Por ejemplo, no escribir esto:

```
mysql> SELECT col_name FROM tbl_name HAVING col_name > 0;
```

Sino esto:

```
mysql> SELECT col_name FROM tbl_name WHERE col_name > 0;
```

- La cláusula *HAVING* se puede referir a funciones agregadas, a las que una cláusula *WHERE* no puede:

```
mysql> SELECT user, MAX(salary) FROM users
->     GROUP BY user HAVING MAX(salary)>10;
```

Sin embargo, esto no funciona en servidores antiguos de MySQL (anteriores a la versión 3.22.5). En su lugar, se puede usar un alias de columna en la lista *SELECT* y referirse al alias en la cláusula *HAVING*:

```
mysql> SELECT user, MAX(salary) AS max_salary FROM users
->     GROUP BY user HAVING max_salary>10;
```

- La cláusula *LIMIT* puede ser usada para limitar a que el número de filas devuelto por la sentencia **SELECT**. *LIMIT* toma uno o dos argumentos numéricos, que deben ser constantes enteras. Con dos argumentos, el primero especifica el desplazamiento de la primera fila a devolver, el segundo especifica el máximo número de filas a devolver. El desplazamiento de la fila inicial es 0 (no 1):

```
mysql> SELECT * FROM table LIMIT 5,10; # Recupera filas 6-15
```

Por compatibilidad con PostgreSQL, MySQL también soporta la sintaxis: *LIMIT row\_count OFFSET offset*. Para recuperar todas las filas a partir de un desplazamiento concreto hasta el final del conjunto de resultados, se puede usar un número muy grande como segundo parámetro. Esta sentencia recupera todas las filas a partir de la 96 hasta el final:

```
mysql> SELECT * FROM table LIMIT 95,18446744073709551615;
```

Con un argumento, el valor especifica el número de filas a devolver desde el principio del conjunto de resultados.

```
mysql> SELECT * FROM table LIMIT 5; # Retrieve first 5 rows
```

En otras palabras, *LIMIT n* equivale a *LIMIT 0,n*.

- El formato *SELECT ... INTO OUTFILE 'file\_name'* de **SELECT** escribe las filas seleccionadas en un fichero. El fichero se crea en el host del servidor, de modo que se debe poseer el privilegio *FILE* para usar esta sintaxis. El fichero no debe existir previamente, entre otras cosas, esto previene que tablas de la base de datos y otros ficheros como `/etc/passwd` puedan ser destruidos. La sentencia *SELECT ... INTO OUTFILE* está pensada para permitir un volcado muy rápido de una tabla en la máquina del servidor. Si se quiere crear el fichero resultado en algún otro host, no se puede usar *SELECT ... INTO OUTFILE*. En ese caso se debe usar en su lugar algún otro programa en el cliente como `mysql -e "SELECT ..." > outfile` en el ordenador cliente para generar el fichero. *SELECT ... INTO OUTFILE* es el complemento de [LOAD DATA INFILE](#); la sintaxis para la parte `export_options` de la sentencia es la misma que para las cláusulas *FIELDS* y *LINES* que se usan con la sentencia [LOAD DATA INFILE](#). *FIELDS ESCAPED BY* controla el modo en que se escriben los caracteres especiales. Si el carácter de *FIELDS ESCAPED BY* no es vacío, se usará como prefijo para los siguientes caracteres en la salida:
  - El carácter *ESCAPED BY*.
  - El carácter *FIELDS [OPTIONALLY] ENCLOSED BY*.
  - El primer carácter de los valores *FIELDS TERMINATED BY* y *LINES TERMINATED BY*.
  - ASCII 0 (que actualmente se escribe seguido del carácter de escape ASCII ``0'`, no un byte de valor cero).

Si el carácter *FIELDS ESCAPED BY* se deja vacío, no se escapa ningún carácter y *NULL* se muestra como *NULL*, no `\N`. Probablemente no sea una buena idea especificar un carácter de escape vacío, sobre todo si existen valores de columnas en los datos que contengan cualquiera de los caracteres de la lista dada. El motivo de esto es que se debe escapar cualquier carácter *FIELDS TERMINATED BY*, *ESCAPED BY* o *LINES TERMINATED BY* para que sea posible leer el fichero más tarde. El carácter ASCII NUL se escapa para que sea más fácil visualizarlo. Como el fichero resultante no tiene que seguir la sintaxis SQL, no es necesario escapar nada más. He aquí un ejemplo para obtener un fichero en formato de valores separados con comas usado por muchos programas:

```
SELECT a,b,a+b INTO OUTFILE "/tmp/result.text"
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY "\n"
FROM test_table;
```

- Si se usa *INTO DUMPFILE* en lugar de *INTO OUTFILE*, MySQL sólo escribirá una fila en el fichero, sin tabulaciones o terminadores y sin realizar ningún proceso de escapado. Esto es práctico si se quiere almacenar un valor *BLOB* en un fichero.
- **Nota:** Cualquier fichero creado por *INTO OUTFILE* y *INTO DUMPFILE* debe tener permiso de escritura para todos los usuarios en el servidor. El motivo es que el servidor MySQL no puede crear un fichero cuyo dueño sea alguien diferente que el usuario que hace la consulta (nunca se debe ejecutar MySQL como root). De modo que el fichero debe tener permiso de escritura para

todo el mundo para que se pueda manejar su contenido.

- Una cláusula *PROCEDURE* nombra a un procedimiento que debe procesar los datos en el conjunto de resultados.
- Si se usa *FOR UPDATE* en un proceso de almacenamiento con bloqueo de página o de filas, las filas examinadas estarán bloqueadas para escritura hasta el final de la operación actual. Usando *IN SHARE MODE* activa el bloqueo de compartir que evita que otras transacciones puedan actualizar o borrar las filas examinadas.

A continuación de la palabra clave **SELECT**, se pueden añadir determinadas opciones que afectan al funcionamiento de la sentencia.

Las opciones *ALL*, *DISTINCT* y *DISTINCTROW* especifican si las filas duplicadas deben ser devueltas. Si no se da ninguna de estas opciones, por defecto se usa *ALL* (se devuelven todas las filas coincidentes). *DISTINCT* y *DISTINCTROW* son sinónimos y especifican que las filas duplicadas en el conjunto de resultados deben ser eliminadas.

*HIGH\_PRIORITY*, *STRAIGHT\_JOIN* y las opciones que empiezan con *SQL\_* son extensiones MySQL al SQL estándar.

- *HIGH\_PRIORITY* dará a **SELECT** mayor prioridad que a sentencias que actualicen una tabla. Sólo se debe usar para consultas que sean muy rápidas y deban ser hechas inmediatamente. Una consulta *SELECT HIGH\_PRIORITY* que se realice mientras la tabla esté bloqueada para lectura se realizará aunque exista una sentencia de actualización que esté esperando a que la table deje de estar bloqueada. *HIGH\_PRIORITY* no se puede usar con sentecias **SELECT** que sean parte de una *UNION*.
- *STRAIGHT\_JOIN* fuerza el optimizador a unir tablas en el orden en que han sido listadas en la cláusula *FROM*. Se puede usar para mejorar la velocidad de una consulta si el optimizador une las tablas en un orden no óptimo. Consultar [EXPLAIN](#). *STRAIGHT\_JOIN* también puede ser usado en la lista de *table\_references*. Ver [JOIN](#)
- *SQL\_BIG\_RESULT* puede ser usada con *GROUP BY* o *DISTINCT* para informar al optimizador que el conjunto resultados puede contener muchas filas. En ese caso, MySQL podrá usar directametne tablas temporales en disco si es necesario. MySQL puede también, en este caso, optar por ordenar una tabla temporal con una clave dentro de los elementos *GROUP BY*.
- *SQL\_BUFFER\_RESULT* fuerza que el resultado sea colocado en una tabla temporal. Esto ayuda a MySQL a librerar bloqueos en tablas más rápidamente y ayuda en casos donde toma mucho tiempo en enviar los datos al cliente.
- *SQL\_SMALL\_RESULT*, puede usarse con *GROUP BY* o *DISTINCT* para informar al optimizador que el resultado será pequeño. En ese caso, MySQL usa tablas temporales rápidas para almecenar la tabla resultado en lugar de usar ordenamiento. A partir de MySQL 3.23 esto normalmente no es necesario.
- *SQL\_CALC\_FOUND\_ROWS* (versión 4.0.0 y siguientes) indica a MySQL que calcule cuántas filas contendrá el conjunto de resultados, ignorando cualquier cláusula *LIMIT*. El número de filas puede recuperarse con *SELECT [FOUND\\_ROWS\(\)](#)*. Con versiones anteriores a la 4.1.0 esto no funcionará junto

con *LIMIT 0*, ya que está optimizado de modo que regrese instantaneamente (resultando un número de filas igual a cero).

- *SQL\_CACHE* dice a MySQL que almacene el resultado de la consulta en un caché si se usa un valor 2 o *DEMAND* para *QUERY\_CACHE\_TYPE*. Para una consulta que use *UNION* o subconsultas, esta opción tendrá efecto al ser usada en cualquier **SELECT** de la consulta.
- *SQL\_NO\_CACHE* indica a MySQL que no almacene el resultado de la consulta en el caché de consulta. Para una consulta que use *UNION* o subconsultas, esta opción tendrá efecto al ser usada en cualquier **SELECT** de la consulta.

(4.1.1)

# SET

```
SET [GLOBAL | SESSION] sql_variable=expression,  
    [[GLOBAL | SESSION] sql_variable=expression] ...
```

**SET** activa varias opciones que afectan al funcionamiento del servidor y el cliente.

Los siguientes ejemplos muestran las diferentes sintaxis que se pueden usar para modificar variables:

En versiones antiguas de MySQL está permitido usar la sintaxis **SET OPTION**, pero esto está desaconsejado ahora.

En MySQL 4.0.3 se han añadido las opciones *GLOBAL* y *SESSION* y acceso a las variables de arranque más importantes.

*LOCAL* se puede usar como sinónimo de *SESSION*.

Si se modifican varias variables en la misma línea de comando, se usa el último modo *GLOBAL* / *SESSION*.

```
SET sort_buffer_size=10000;  
SET @@local.sort_buffer_size=10000;  
SET GLOBAL sort_buffer_size=1000000, SESSION sort_buffer_size=1000000;  
SET @@sort_buffer_size=1000000;  
SET @@global.sort_buffer_size=1000000, @@local.sort_buffer_size=1000000;
```

La sintaxis *@@variable\_name* se soporta para hacer la sintaxis de MySQL compatible con otras bases de datos.

Las variables de sistema que se pueden modificar se describen en la sección de variables de sistema de este manual.

Si se está usando *SESSION* (por defecto), la opción modificada permanece en efecto hasta que la sesión actual termine, o hasta que se cambie la opción a un valor diferente. Si se usa *GLOBAL*, que requiere el privilegio *SUPER*, la opción se recuerda y se usa para nuevas conexiones hasta que el servidor se reinicie. Si se quiere que una opción se modifique permanentemente, se debe activar en un fichero de opciones.

Para impedir el uso incorrecto, MySQL producirá un error si se usa **SET GLOBAL** con una variable que sólo se puede usar con **SET SESSION** o si no se usa **SET GLOBAL** con una variable global.

Si se quiere cambiar una variable de *SESSION* a un valor *GLOBAL* o un valor *GLOBAL* a su valor por defecto de MySQL, se puede asignar *DEFAULT*.

```
SET max_join_size=DEFAULT;
```

Esto es idéntico a:

```
SET @@session.max_join_size=@@global.max_join_size;
```

Si se quiere restringir el valor máximo al que una variable del servidor puede ser asignada, se puede especificar ese máximo usando la opción de línea de comando **--maximum-variable-name**.

Se puede obtener una lista de la mayoría de las variables con [SHOW VARIABLES](#). Se puede obtener el valor de una variable específica con la sintaxis @@[global.|local.]variable\_name:

```
SHOW VARIABLES like "max_join_size";  
SHOW GLOBAL VARIABLES like "max_join_size";  
SELECT @@max_join_size, @@global.max_join_size;
```

A continuación se expone una descripción de las variables que usan una sintaxis no estándar de **SET** y algunas de las otras variables. El resto de las definiciones de variables pueden encontrarse en la sección de variables del sistema, entre las opciones de arranque o en la descripción de [SHOW VARIABLES](#).

**AUTOCOMMIT= 0 | 1**

Si se asigna 1, todos los cambios de una tabla se hacen inmediatamente. Para comenzar una transacción multicomando, se debe usar la sentencia [BEGIN](#). Ver también las sintaxis de [START TRANSACTION](#), [COMMIT](#) y [ROLLBACK](#). Si se asigna 0 se debe usar [COMMIT](#) para aceptar la transacción o [ROLLBACK](#) para cancelarla. Cuando se cambia el modo *AUTOCOMMIT* de 0 a 1, MySQL realiza un [COMMIT](#) automático de cualquier transacción abierta.

**BIG\_TABLES = 0 | 1**

Se se asigna 1, todas las tablas temporales se almacenan en disco en lugar de en memoria. Esto puede ser un poco más lento, pero no se obtendrá el error "The table tbl\_name is full" para operaciones **SELECT** grandes que requieren una tabla temporal grande. El valor por defecto para nuevas conexiones es 0 (es decir, usar tablas temporales en memoria). Esta variable fue llamada previamente *SQL\_BIG\_TABLES*. En MySQL 4.0, normalmente no será necesario modificar esta variable, ya que MySQL convierte tablas en memoria a tablas en disco de forma automática, cuando es necesario.

**CHARACTER SET** character\_set\_name | DEFAULT

Esto mapea todas las cadenas desde y hacia el cliente con el mapeado dado. Actualmente la única opción para *character\_set\_name* es "cp1251\_koi8", pero se pueden añadir fácilmente nuevos mapeados mediante la edición del fichero 'sql/convert.cc' en la distribución fuente de MySQL. El mapeado por defecto puede ser restaurado usando el valor *DEFAULT* para *character\_set\_name*. Puede verse que la sintaxis para cambiar la opción **CHARACTER SET** difiere de la sintaxis para modificar otras opciones.

**INSERT\_ID** = #

Cambia el valor a usar en el siguiente comando **INSERT** o **ALTER TABLE** cuando se inserte un valor *AUTO\_INCREMENT*. Esto se usa principalmente con el diario binario.

**LAST\_INSERT\_ID** = #

Cambia el valor que retornará desde **LAST\_INSERT\_ID()**. Este se almacena en el diario binario cuando se usa **LAST\_INSERT\_ID()** en un comando que actualice una tabla.

**LOW\_PRIORITY\_UPDATES** = 0 | 1

Si se asigna 1, todas las sentencias **INSERT**, **UPDATE**, **DELETE** y **LOCK TABLE WRITE** esperan hasta que no haya sentencias **SELECT** o **LOCK TABLE READ** pendientes en la tabla afectada. Esta variable previamente se llamaba *SQL\_LOW\_PRIORITY\_UPDATES*.

**MAX\_JOIN\_SIZE** = value | DEFAULT

No admite sentencias **SELECT** que probablemente necesiten examinar más de 'value' combinaciones de valor de fila o es probable que haga más 'value' accesos a disco. Mediante este valor se pueden detener sentencias **SELECT** donde las claves no se han usado apropiadamente y que es probable que requieran mucho tiempo. Asignando un valor diferente de *DEFAULT* asigna 0 a *SQL\_BIG\_SELECTS*. Si se activa el valor de

*SQL\_BIG\_SELECTS* otra vez, la variable *SQL\_MAX\_JOIN\_SIZE* será ignorada. Se puede asignar un valor por defecto para esta variable arrancando mysqld con la opción `--max_join_size=value`. Esta variable se llamó previamente *SQL\_MAX\_JOIN\_SIZE*. Si un resultado de una consulta ya está en el caché de consultas, no se realiza ninguna comprobación de tamaño, porque el resultado ya ha sido calculado y no se cargará al servidor para enviarlo al cliente.

**PASSWORD = PASSWORD('some password')**

Asigna la contraseña para el usuario actual. Cualquier usuario no anónimo puede modificar su propia contraseña.

**PASSWORD FOR user = PASSWORD('some password')**

Asigna la contraseña para un usuario específico en el ordenador servidor actual. Sólo un usuario con acceso a la base de datos mysql puede hacer esto. El usuario puede darse en el formato `user@hostname`, donde el 'user' y el 'hostname' son exactamente como aparecen listados en las columnas 'User' y 'Host' de la tabla 'mysql.user'. Por ejemplo, si se tiene una entrada con valores para los campos 'User' y 'Host' de 'bob' and '%.loc.gov', se debe escribir:

```
mysql> SET PASSWORD FOR 'bob'@'%.loc.gov' = PASSWORD('newpass');
```

Que es equivalente a:

```
mysql> UPDATE mysql.user SET Password=PASSWORD('newpass')
-> WHERE User='bob' AND Host='%.loc.gov';
mysql> FLUSH PRIVILEGES;
```

**QUERY\_CACHE\_TYPE = OFF | ON | DEMAND**

**QUERY\_CACHE\_TYPE = 0 | 1 | 2**

Modifica la configuración del caché de consultas para este hilo.

Opción	Descripción
0 o OFF	No hay caché para los resultados recuperados.
1 o ON	Se almacenan todos los resultados excepto consultas <a href="#">SELECT SQL_NO_CACHE ...</a>

2 o DEMAND	Se almacenan sólo consultas <u>SELECT SQL_CACHE ...</u> .
------------	---

SQL\_AUTO\_IS\_NULL = 0 | 1

Si se asigna 1 (por defecto), se puede encontrar la última fila insertada en una tabla que contenga una columna *AUTO\_INCREMENT* usando la siguiente construcción: *WHERE auto\_increment\_column IS NULL*. Esto se usa por algunos programas **ODBC** como **Access**.

SQL\_BIG\_SELECTS = 0 | 1

Si se asigna 0, MySQL aborta las sentencias SELECT que probablemente requieran mucho tiempo (es decir, sentencias para las que el optimizador estima que el número de filas a examinar excederá el valor de *MAX\_JOIN\_SIZE*). Esto es frecuente cuando se usa una sentencia *WHERE* poco aconsejable. El valor por defecto para una conexión nueva es 1, esto permite todas las sentencias SELECT. Si se asigna a *MAX\_JOIN\_SIZE* un valor que no sea *DEFAULT*, *SQL\_BIG\_SELECTS* tomará el valor 0.

SQL\_BUFFER\_RESULT = 0 | 1

*SQL\_BUFFER\_RESULT* fuerza que los resultados de las sentencias SELECT se almacenen en tablas temporales. Esto ayuda a MySQL a liberar bloqueos de tablas más pronto y ayuda en casos donde toma mucho tiempo enviar el conjunto de resultados al cliente.

SQL\_LOG\_BIN = 0 | 1

Si se asigna 0, no se realiza diario para el cliente en el diario binario, si el cliente tiene el privilegio *SUPER*.

SQL\_LOG\_OFF = 0 | 1

Si se asigna 1, no se actualiza el diario estándar para este cliente, si el cliente tiene el privilegio *SUPER*.

SQL\_LOG\_UPDATE = 0 | 1

Si se asigna 0, no se actualiza el diario de actualización para el cliente, si el cliente tiene el privilegio *SUPER*. Esta variable está desaconsejada desde la versión 5.0.

SQL\_QUOTE\_SHOW\_CREATE = 0 | 1

Si se asigna 1, [SHOW CREATE TABLE](#) entrecomilla los nombres de la tabla y de las columnas. Por defecto vale 1, de modo que el replicado de tablas funciona con cualquier nombre de campo.

SQL\_SAFE\_UPDATES = 0 | 1

Si se asigna 1, MySQL aborta sentencias [UPDATE](#) o [DELETE](#) que no usen una clave o *LIMIT* en la cláusula *WHERE*. Esto hace posible detener actualizaciones equivocadas cuando se crean sentencias SQL manualmente.

SQL\_SELECT\_LIMIT = value | DEFAULT

El máximo número de registros retornados por sentencias [SELECT](#). Si un [SELECT](#) tiene una cláusula *LIMIT*, este valor tiene preferencia sobre el valor de *SQL\_SELECT\_LIMIT*. El valor por defecto para una nueva conexión es "unlimited". Si se ha modificado el límite, es posible restaurar el valor por defecto especificando el valor *DEFAULT* en *SQL\_SELECT\_LIMIT*.

TIMESTAMP = timestamp\_value | DEFAULT

Cambia la hora para este cliente. Esto se usa para recuperar la hora original si se usa el diario binario para restaurar filas. *timestamp\_value* debe tener el formato Unix 'epoch timestamp', no el de MySQL.

(4.0)

# SET TRANSACTION

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL  
{ READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }
```

Asigna el nivel de aislamiento de transacción para transacciones globales, la sesión completa o para la siguiente transacción.

El comportamiento por defecto es asignar el nivel de aislamiento para la siguiente transacción (no comenzada). Si se usa *GLOBAL*, la sentencia asigna el nivel de transacción global por defecto para todas las nuevas conexiones creadas desde ese punto (pero no para las conexiones existentes). Se necesita el privilegio *SUPER* para hacer esto. Usando la opción *SESSION* se asigna el nivel de transacción por defecto para todas las futuras transacciones realizadas en la conexión actual.

**InnoDB** soporta cada uno de estos niveles desde MySQL 4.0.5. El nivel por defecto es *REPEATABLE READ*.

Se puede asignar el nivel de aislamiento global por defecto para *mysqld* con *--transaction-isolation=...*

(4.0)

# SHOW

```
SHOW [FULL] COLUMNS FROM tbl_name [FROM db_name] [LIKE 'patrón']  
SHOW CREATE DATABASE db_name  
SHOW CREATE TABLE tbl_name  
SHOW DATABASES [LIKE 'pattern']  
SHOW [STORAGE] ENGINES  
SHOW ERRORS [LIMIT [offset,] row_count]  
SHOW GRANTS FOR user  
SHOW INDEX FROM tbl_name [FROM db_name]  
SHOW INNODB STATUS  
SHOW [BDB] LOGS  
SHOW PRIVILEGES  
SHOW [FULL] PROCESSLIST  
SHOW STATUS [LIKE 'pattern']  
SHOW TABLE STATUS [FROM db_name] [LIKE 'pattern']  
SHOW [OPEN] TABLES [FROM db_name] [LIKE 'pattern']  
SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'pattern']  
SHOW WARNINGS [LIMIT [offset,] row_count]
```

**SHOW** proporciona información sobre bases de datos, tablas, columnas o información de estado sobre el servidor. Si se usa la parte *LIKE*, la cadena de patrón puede usar los caracteres '%' y '\_' como comodines. El patrón es útil para restringir la salida de la sentencia a los valores que coincidan con él.

Existen otros formatos de estas sentencias:

La sentencia **SHOW** tiene formatos que proporcionan información sobre replicado de servidores maestros y esclavos:

```
SHOW BINLOG EVENTS  
SHOW MASTER LOGS  
SHOW MASTER STATUS  
SHOW SLAVE HOSTS  
SHOW SLAVE STATUS
```

(4.1.1)

# SHOW CHARACTER SET

```
SHOW CHARACTER SET [LIKE 'pattern']
```

La sentencia **SHOW CHARACTER SET** muestra los conjuntos de caracteres disponibles. Puede tener una cláusula opcional *LIKE* que indique qué nombres de conjuntos de caracteres buscar. Por ejemplo:

```
mysql> SHOW CHARACTER SET LIKE 'latin%';
```

Charset	Description	Default collation	Maxlen
latin1	ISO 8859-1 West European	latin1_swedish_ci	1
latin2	ISO 8859-2 Central European	latin2_general_ci	1
latin5	ISO 8859-9 Turkish	latin5_turkish_ci	1
latin7	ISO 8859-13 Baltic	latin7_general_ci	1

La columna *Maxlen* muestra el número máximo de bytes usados para almacenar un carácter.

**SHOW CHARACTER SET** está disponible a partir de MySQL 4.1.0.

(4.1.1)

# SHOW COLLATION

```
SHOW COLLATION [LIKE 'pattern']
```

La salida de **SHOW COLLATION** incluye todos los conjuntos de caracteres disponibles. La cláusula opcional *LIKE* indica qué nombres de conjuntos de reglas buscar. Por ejemplo:

```
mysql> SHOW COLLATION LIKE 'latin1%';
```

Collation	Charset	Id	Default	Compiled	Sortlen
latin1_german1_ci	latin1	5			0
latin1_swedish_ci	latin1	8	Yes	Yes	0
latin1_danish_ci	latin1	15			0
latin1_german2_ci	latin1	31		Yes	2
latin1_bin	latin1	47		Yes	0
latin1_general_ci	latin1	48			0
latin1_general_cs	latin1	49			0
latin1_spanish_ci	latin1	94			0

La columna *Default* indica si un conjunto de reglas (collation) es el usado por defecto para un conjunto de caracteres. *Compiled* indica si el conjunto de caracteres está compilado en el servidor. *Sortlen* está relacionado con la cantidad de memoria requerida para ordenar cadenas expresadas en el conjunto de caracteres.

**SHOW COLLATION** está disponible a partir de MySQL 4.1.0.

(4.1.1)

# SHOW COLUMNS

## SHOW FIELDS

```
SHOW [FULL] COLUMNS FROM tbl_name [FROM db_name] [LIKE 'patrón']
```

**SHOW COLUMNS** lista las columnas de una tabla dada. Si los tipos de columna difieren de los que se esperaba a partir de la sentencia **CREATE TABLE** usada, hay que tener en cuenta que a veces MySQL cambia tipos de columnas cuando se crea o altera una tabla. Las condiciones para que esto ocurra se describen al final de **CREATE TABLE**.

La palabra clave *FULL* se puede usar a partir de MySQL 3.23.32. Hace que la salida incluya los privilegios que se poseen para cada columna. A partir de MySQL 4.1, *FULL* también hace que se muestre cualquier comentario por columna que exista.

Se puede usar *db\_name.tbl\_name* como una alternativa a la sintaxis *tbl\_name FROM db\_name syntax*. Estas dos sentencias son equivalentes:

```
mysql> SHOW COLUMNS FROM mytable FROM mydb;  
mysql> SHOW COLUMNS FROM mydb.mytable;
```

**SHOW FIELDS** es un sinónimo de **SHOW COLUMNS**. También se pueden listar las columnas de una tabla con el comando *mysqlshow db\_name tbl\_name*.

La sentencia **DESCRIBE** proporciona información similar a **SHOW COLUMNS**.

(4.1.1)

# SHOW CREATE DATABASE

```
SHOW CREATE {DATABASE | SCHEMA} db_name
```

Muestra una sentencia [CREATE DATABASE](#) que creará la base de datos dada. Fue añadida en MySQL 4.1. **SHOW CREATE SCHEMA** puede ser usada a partir de MySQL 5.0.2.

```
mysql> SHOW CREATE DATABASE test\G
***** 1. row *****
      Database: test
Create Database: CREATE DATABASE `test`
                /*!40100 DEFAULT CHARACTER SET latin1 */
```

(4.1.1)

# SHOW CREATE TABLE

```
SHOW CREATE TABLE tbl_name
```

Muestra la sentencia [CREATE TABLE](#) que creará la tabla dada, fue añadida en MySQL 3.23.20.

```
mysql> SHOW CREATE TABLE t\G
***** 1. row *****
      Table: t
Create Table: CREATE TABLE t (
  id INT(11) default NULL auto_increment,
  s char(60) default NULL,
  PRIMARY KEY (id)
) TYPE=MyISAM
```

**SHOW CREATE TABLE** entrecomilla los nombres de tabla y columnas de acuerdo con el valor de la opción `SQL_QUOTE_SHOW_CREATE`. Ver sintaxis de [SET](#).

(4.1.1)

# SHOW CREATE VIEW

```
SHOW CREATE VIEW view_name
```

Esta sentencia muestra una sentencia [CREATE VIEW](#) que creará la vista dada.

```
mysql> SHOW CREATE VIEW v;
+-----+-----+
| Table | Create Table |
+-----+-----+
| v     | CREATE VIEW `test`.`v` AS select 1 AS `a`,2 AS `b` |
+-----+-----+
```

Esta sentencia se añadió en MySQL 5.0.1.

(5.0.1)

# SHOW DATABASES

```
SHOW {DATABASES | SCHEMAS} [LIKE 'patrón']
```

**SHOW DATABASES** lista las bases de datos en el ordenador del servidor **MySQL**. También se puede obtener esa lista usando el comando *mysqlshow*. Desde **MySQL 4.0.2**, sólo es posible ver aquellas bases de datos para las cuales se dispone algún tipo de privilegio, si no se posee el privilegio global *SHOW DATABASES*.

Si el servidor fue arrancado con la opción *--skip-show-database*, no se podrá usar esta sentencia de ninguna manera, salvo que se disponga del privilegio *SHOW DATABASES*.

**SHOW SCHEMAS** se puede usar desde **MySQL 5.0.2**.

(4.1.1)

# SHOW ENGINES

```
SHOW [STORAGE] ENGINES
```

**SHOW ENGINES** muestra la información de estado sobre los motores de almacenamiento. Esto es particularmente útil para comprobar si un motor de almacenamiento está soportado, o para ver cual es el motor de almacenamiento por defecto. Esta sentencia está implementada a partir de MySQL 4.1.2. **SHOW TABLE TYPES** es un sinónimo desaconsejado.

```
mysql> SHOW ENGINES\G
***** 1. row *****
Engine: MyISAM
Support: DEFAULT
Comment: Default engine as of MySQL 3.23 with great performance
***** 2. row *****
Engine: HEAP
Support: YES
Comment: Alias for MEMORY
***** 3. row *****
Engine: MEMORY
Support: YES
Comment: Hash based, stored in memory, useful for temporary tables
***** 4. row *****
Engine: MERGE
Support: YES
Comment: Collection of identical MyISAM tables
***** 5. row *****
Engine: MRG_MYISAM
Support: YES
Comment: Alias for MERGE
***** 6. row *****
Engine: ISAM
Support: NO
Comment: Obsolete storage engine, now replaced by MyISAM
***** 7. row *****
Engine: MRG_ISAM
Support: NO
Comment: Obsolete storage engine, now replaced by MERGE
***** 8. row *****
Engine: InnoDB
Support: YES
Comment: Supports transactions, row-level locking, and foreign keys
```

## SHOW ENGINES

```
***** 9. row *****
Engine: INNODB
Support: YES
Comment: Alias for INNODB
***** 10. row *****
Engine: BDB
Support: YES
Comment: Supports transactions and page-level locking
***** 11. row *****
Engine: BERKELEYDB
Support: YES
Comment: Alias for BDB
***** 12. row *****
Engine: NDBCLUSTER
Support: YES
Comment: Clustered, fault-tolerant, memory-based tables
***** 13. row *****
Engine: NDB
Support: YES
Comment: Alias for NDBCLUSTER
***** 14. row *****
Engine: EXAMPLE
Support: YES
Comment: Example storage engine
***** 15. row *****
Engine: ARCHIVE
Support: YES
Comment: Archive storage engine
***** 16. row *****
Engine: CSV
Support: YES
Comment: CSV storage engine
***** 17. row *****
Engine: FEDERATED
Support: YES
Comment: Federated MySQL storage engine
```

El valor *Support* indica si un motor de almacenamiento en particular está soportado, y cual es el motor por defecto. Por ejemplo, si el servidor se arranca con la opción `--default-table-type=InnoDB`, entonces el valor *Support* para la fila **InnoDB** será el valor *DEFAULT*.

(4.1.1)

# SHOW ERRORS

```
SHOW ERRORS [LIMIT [offset,] row_count]
SHOW COUNT(*) ERRORS
```

Esta sentencia es similar a [SHOW WARNINGS](#), excepto que en lugar de mostrar errores, avisos y notas, muestra sólo errores. **SHOW ERRORS** está disponible desde MySQL 4.1.0.

La cláusula *LIMIT* tiene la misma sintaxis que para la sentencia [SELECT](#).

La sentencia **SHOW COUNT(\*) ERRORS** muestra el número de errores. También se puede recuperar este número desde la variable *the error\_count*:

```
SHOW COUNT(*) ERRORS;
SELECT @@error_count;
```

Para más información ver la sintaxis de [SHOW WARNINGS](#).

(4.1.1)

# SHOW GRANTS

```
SHOW GRANTS FOR user
```

Esta sentencia lista las sentencias GRANT que deben ser lanzadas para duplicar los privilegios para una cuenta de usuario MySQL.

```
mysql> SHOW GRANTS FOR 'root'@'localhost';
+-----+-----+
| Grants for root@localhost |
+-----+-----+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION |
+-----+-----+
```

Desde MySQL 4.1.2, para listar los privilegios para la sesión actual, se puede usar cualquiera de las siguientes sentencias:

```
SHOW GRANTS ;
SHOW GRANTS FOR CURRENT_USER ;
SHOW GRANTS FOR CURRENT_USER ( ) ;
```

Antes de MySQL 4.1.2, se puede averiguar qué usuario fue autenticado para la sesión seleccionando el valor de la función CURRENT\_USER() (nueva en MySQL 4.0.6). Entonces se usa ese valor en la sentencia **SHOW GRANTS**.

**SHOW GRANTS** está disponible a partir de MySQL 3.23.4.

(4.1.1)

# SHOW INDEX

## SHOW KEYS

```
SHOW INDEX FROM tbl_name [FROM db_name]
```

**SHOW INDEX** devuelve la información de índices de una tabla en un formato parecido a la llamada *SQLStatistics* de **ODBC**.

SHOW INDEX devuelve los siguientes campos:

Campo	Descripción
Table	El nombre de la tabla.
Non_unique	0 si el índice no puede tener duplicados, 1 si puede.
Key_name	El nombre del índice.
Seq_in_index	El número de secuencia de columna del índice, empezando en 1.
Column_name	El nombre de columna.
Collation	El modo en que la columna se ordena en el índice. En MySQL, puede tener los valores 'A' (Ascending) o NULL (no ordenado).
Cardinality	El número de valores únicos en el índice. Este valor se actualiza mediante la ejecución de la <a href="#">ANALYZE TABLE</a> o <i>myisamchk -a</i> . La cardinalidad se cuenta en base a las estadísticas almacenadas como enteros, de modo que no es necesario hacer aproximaciones para tablas pequeñas.
Sub_part	El número de caracteres indexados si la columna está indexada parcialmente. NULL si se indexa la columna completa.
Packed	Indica el modo en que se empaqueta la clave. NULL si no se empaqueta.
Null	Contiene <i>YES</i> si la columna puede contener <i>NULL</i> , " si no.
Index_type	El método de índice usado (BTREE, FULLTEXT, HASH, RTREE).
Comment	Varios comentarios. Antes de MySQL 4.0.2 cuando se añadió la columna <i>Index_type</i> , <i>Comment</i> indica si un índice es <i>FULLTEXT</i> .

Las columnas *Packed* y *Comment* se añadieron en MySQL 3.23.0. Las columnas *Null* y *Index\_type* en MySQL 4.0.2.

Se puede usar *db\_name.tbl\_name* como alternativa a la sintaxis *tbl\_name FROM db\_name*. Estas dos

sentencias son equivalentes:

```
mysql> SHOW INDEX FROM mytable FROM mydb;  
mysql> SHOW INDEX FROM mydb.mytable;
```

**SHOW KEYS** es un sinónimo de **SHOW INDEX**. También se pueden listar los índices de tablas con el comando *mysqlshow -k db\_name tbl\_name*.

(4.1.1)

# ***SHOW INNODB STATUS***

```
SHOW INNODB STATUS
```

Esta sentencia muestra información detallada sobre el estado del motor de almacenamiento **InnoDB**.

(4.1.1)

# SHOW LOGS

```
SHOW [BDB] LOGS
```

**SHOW LOGS** muestra información de estado sobre los ficheros de diario existentes. Se implementó en MySQL 3.23.29. Actualmente, sólo muestra información sobre los ficheros de diario **Berkeley DB**, así como su alias (disponible desde MySQL 4.1.1) que es **SHOW BDB LOGS**.

**SHOW LOGS** devuelve los siguientes campos:

File	El camino completo para el fichero de diario
Type	El tipo de fichero de diario ( <i>BDB</i> para ficheros de diario <b>Berkeley DB</b> ).
Status	El estado del fichero de diario ( <i>FREE</i> si el fichero puede ser borrado, o <i>IN USE</i> si el fichero es necesario para el subsistema de transacción).

(4.1.1)

# SHOW PRIVILEGES

## SHOW PRIVILEGES

**SHOW PRIVILEGES** muestra la lista de privilegios del sistema que el servidor MySQL actual soporta. Esta sentencia se implementa desde MySQL 4.1.0.

```
mysql> SHOW PRIVILEGES\G
***** 1. row *****
Privilege: Select
Context: Tables
Comment: To retrieve rows from table
***** 2. row *****
Privilege: Insert
Context: Tables
Comment: To insert data into tables
***** 3. row *****
Privilege: Update
Context: Tables
Comment: To update existing rows
***** 4. row *****
Privilege: Delete
Context: Tables
Comment: To delete existing rows
***** 5. row *****
Privilege: Index
Context: Tables
Comment: To create or drop indexes
***** 6. row *****
Privilege: Alter
Context: Tables
Comment: To alter the table
***** 7. row *****
Privilege: Create
Context: Databases,Tables,Indexes
Comment: To create new databases and tables
***** 8. row *****
Privilege: Drop
Context: Databases,Tables
Comment: To drop databases and tables
***** 9. row *****
Privilege: Grant
Context: Databases,Tables
```

## SHOW PRIVILEGES

```
Comment: To give to other users those privileges you possess
***** 10. row *****
Privilege: References
Context: Databases,Tables
Comment: To have references on tables
***** 11. row *****
Privilege: Reload
Context: Server Admin
Comment: To reload or refresh tables, logs and privileges
***** 12. row *****
Privilege: Shutdown
Context: Server Admin
Comment: To shutdown the server
***** 13. row *****
Privilege: Process
Context: Server Admin
Comment: To view the plain text of currently executing queries
***** 14. row *****
Privilege: File
Context: File access on server
Comment: To read and write files on the server
```

(4.1.1)

# SHOW PROCESSLIST

```
SHOW [FULL] PROCESSLIST
```

**SHOW PROCESSLIST** muestra qué procesos están corriendo. También se puede obtener esta información usando el comando *mysqladmin processlist*. Si se posee el privilegio *SUPER*, se pueden ver todos los procesos. En caso contrario, sólo será posible ver los propios procesos (esto es, los procesos asociados con la cuenta MySQL que se está usando). Ver la sintaxis de [KILL](#). Si no se usa la palabra clave *FULL*, sólo se muestran los 100 primeros caracteres de cada consulta.

A partir de MySQL 4.0.12, la sentencia informa sobre el nombre de la máquina para conexiones TCP/IP usando el formato `host_name:client_port` para hacer más sencillo determinar qué cliente está haciendo cada cosa.

Esta sentencia es muy práctica si se obtiene el mensaje de error "too many connections" y se quiere averiguar qué está pasando. MySQL reserva una conexión extra para ser usada por cuentas que tengan el privilegio *SUPER*, para asegurar que los administradores siempre tendrán la posibilidad de conectar y verificar el sistema (asumiendo que no se ha dado tal privilegio a todos los usuarios).

Algunos estados frecuentes en la salida de **SHOW PROCESSLIST** son:

Checking table	El proceso está realizando una comprobación (automática) de la tabla.
Closing tables	Significa que el proceso está enviando los datos modificados de la tabla al disco y cerrando las tablas usadas. Esto debe ser una operación rápida. Si no lo es, entonces se debe verificar que el disco no esté lleno y que el disco no está siendo muy usado.
Connect Out	Esclavo conectando al maestro.
Copying to tmp table on disk	El conjunto de resultados temporal era más grande que <i>tmp_table_size</i> y el proceso está cambiando la tabla temporal del formato en memoria al basado en disco, para ahorrar memoria.
Creating tmp table	El proceso está creando una tabla temporal para almacenar una parte del resultado de una consulta.
deleting from main table	El servidor está ejecutando la primera parte de un borrado multitabla y borrando sólo desde la primera tabla.
deleting from reference tables	El servidor está ejecutando la segunda parte de un borrado multitabla y eliminando las filas coincidentes de otras tablas.

Flushing tables	El proceso está ejecutando <b>FLUSH TABLES</b> y esperando a que todos los procesos cierren sus tablas.
Killed	Alguien ha enviado un 'kill' al proceso y se abortará la vez siguiente que verifique la marca de 'kill'. La marca se verifica en cada bucle exterior en MySQL, pero en algunos casos podrá requerir un pequeño tiempo que el proceso se elimine. Si el proceso está bloqueado por otro, la finalización tendrá efecto tan pronto como el otro proceso retire el bloqueo.
Locked	La consulta está bloqueada por otra consulta.
Sending data	Esta procesando filas para una sentencia <b>SELECT</b> y también está enviando datos al cliente.
Sorting for group	El proceso está haciendo un ordenamiento para satisfacer un <i>GROUP BY</i> .
Sorting for order	El proceso está haciendo un ordenamiento para satisfacer un <i>ORDER BY</i> .
Opening tables	El proceso está intentado abrir una tabla. Este debe ser un procedimiento muy rápido, a no ser que algo impida la apertura. Por ejemplo, una sentencia <b>ALTER TABLE</b> o una sentencia <b>LOCK TABLES</b> pueden impedir la apertura de una tabla hasta que la sentencia haya terminado.
Removing duplicates	La consulta ha usado <b>SELECT DISTINCT</b> de tal modo que MySQL no puede optimizar la operación de distinción en una primera etapa. Debido a esto, MySQL necesita una etapa extra para eliminar todas las filas duplicadas antes de enviar el resultado al cliente.
Reopen table	El proceso ha obtenido un bloqueo para la tabla, pero se ha notificado que después de obtenerlo la estructura de la tabla ha cambiado. Ha liberado el bloqueo, cerrado la tabla y ahora está intentando reabirla.
Repair by sorting	El código de reparado está usando un ordenamiento para crear los índices.
Repair with keycache	El código de reparado está usando la creación de claves una a una a través del caché de claves. Esto es mucho más lento que 'Repair by sorting'.
Searching rows for update	El proceso está ejecutando una primera fase para encontrar todas las filas coincidentes antes de actualizarlas. Esto tiene que hacerse si el <b>UPDATE</b> está modificando el índice que se usa para encontrar las filas involucradas.
Sleeping	El proceso está esperando a que el cliente le envíe una nueva sentencia.

System lock	El proceso está esperando a obtener un bloque de sistema externo para la tabla. Si no se están usando varios servidores mysqld que estén accediendo a las mismas tablas, se pueden desactivar los bloqueos de sistema con la opción <code>--skip-external-locking</code> .
Upgrading lock	El manipulador de <a href="#">INSERT DELAYED</a> está intentando obtener un bloqueo para la tabla para insertar filas.
Updating	El proceso está buscando filas para actualizar y actualizándolas.
User Lock	El proceso está esperando un <a href="#">GET_LOCK()</a> .
Waiting for tables	El proceso tiene una notificación de que la estructura para una tabla subyacente ha cambiado y necesita reabrir la tabla para obtener la nueva estructura. Sin embargo, para que pueda reabrir la tabla, debe esperar hasta que otros procesos hayan cerrado la tabla en cuestión. Esta notificación se produce si otro proceso ha usado <a href="#">FLUSH TABLES</a> o una de las siguientes sentencias para la tabla en cuestión: <a href="#">FLUSH TABLES tbl_name</a> , <a href="#">ALTER TABLE</a> , <a href="#">RENAME TABLE</a> , <a href="#">REPAIR TABLE</a> , <a href="#">ANALYZE TABLE</a> u <a href="#">OPTIMIZE TABLE</a> .
waiting for handler insert	El manipulador de <a href="#">INSERT DELAYED</a> ha procesado todas las inserciones pendientes y está esperando por más.

Muchos estados corresponden con operaciones muy rápidas. Si un proceso permanece en cualquiera de esos estados por muchos segundos, probablemente existe un problema que necesita ser investigado.

Hay otros estados que no se mencionan en la lista anterior, pero muchos de ellos son útiles sólo para encontrar bugs en el servidor.

(4.1.1)

# SHOW STATUS

```
SHOW STATUS [LIKE 'pattern']
```

**SHOW STATUS** proporciona información sobre el estado del servidor. Esta información también se puede obtener usando el comando *mysqladmin extended-status*.

A continuación se muestra una salida parcial. La lista de variables y sus valores pueden ser diferentes para cada servidor. El significado de cada variable se muestra en la sección 5.2.4 "Server Status Variables" del manual original de MySQL.

```
mysql> SHOW STATUS;
+-----+-----+
| Variable_name          | Value          |
+-----+-----+
| Aborted_clients        | 0              |
| Aborted_connects      | 0              |
| Bytes_received         | 155372598     |
| Bytes_sent             | 1176560426    |
| Connections            | 30023          |
| Created_tmp_disk_tables| 0              |
| Created_tmp_tables     | 8340           |
| Created_tmp_files      | 60             |
| ...                    |                |
| Open_tables            | 1              |
| Open_files             | 2              |
| Open_streams           | 0              |
| Opened_tables          | 44600          |
| Questions              | 2026873        |
| ...                    |                |
| Table_locks_immediate  | 1920382        |
| Table_locks_waited     | 0              |
| Threads_cached         | 0              |
| Threads_created        | 30022          |
| Threads_connected      | 1              |
| Threads_running        | 1              |
| Uptime                 | 80380          |
+-----+-----+
```

Con la cláusula *LIKE*, la sentencia muestra sólo aquellas variables que coincidan con el patrón:

```
mysql> SHOW STATUS LIKE 'Key%';
+-----+-----+
| Variable_name      | Value          |
+-----+-----+
| Key_blocks_used    | 14955          |
| Key_read_requests  | 96854827       |
| Key_reads          | 162040         |
| Key_write_requests | 7589728        |
| Key_writes         | 3813196        |
+-----+-----+
```

(4.1.1)

# SHOW TABLE STATUS

```
SHOW TABLE STATUS [FROM db_name] [LIKE 'pattern']
```

**SHOW TABLE STATUS** funciona como [SHOW TABLE](#), pero proporciona mucha más información sobre cada tabla. También se puede obtener esta lista usando el comando `mysqlshow --status db_name`. Esta sentencia se añadió en MySQL 3.23. Desde MySQL 5.0.1, también muestra información sobre vistas.

**SHOW TABLE STATUS** devuelve los siguientes campos:

Name	El nombre de la tabla.
Engine	El motor de almacenamiento usado para la tabla. Antes de MySQL 4.1.2, este valor se etiquetaba como 'Type'.
Version	El número de versión del fichero '.frm' de la tabla.
Row_format	El formato de almacenamiento de filas (Fijod, Dinámico o comprimido).
Rows	El número de filas. Algunos motores de almacenamiento, como <b>MyISAM</b> e <b>ISAM</b> , almacenan el contador exacto. Para otros motores de almacenamiento, como <b>InnoDB</b> , este valor es una aproximación, y puede diferir del valor actual hasta un 40 a 50%. En esos casos, usar <a href="#">SELECT COUNT(*)</a> para obtener un contador preciso.
Avg_row_length	La longitud media de fila.
Data_length	La longitud del fichero de datos.
Max_data_length	La longitud máxima del fichero de datos. Para formatos de filas de longitud fija, este es el número máximo de filas en la tabla. Para formatos dinámicos, es el número total de bytes de datos que puede almacenar la tabla, dado el tamaño del puntero de datos usado.
Index_length	la longitud del fichero de índices.
Data_free	Número de bytes reservados pero no usados.
Auto_increment	El siguiente valor <i>AUTO_INCREMENT</i> .
Create_time	Cunado fue creada la tabla.
Update_time	Cuando fue actualizado el fichero de datos por última vez.
Check_time	Cuando se comprobó la tabla por última vez.
Collation	Conjunto de caracteres y reglas para la tabla. (Nuevo en 4.1.1)

Checksum	El valor de checksum actual (si existe). (Nuevo en 4.1.1)
Create_options	Opciones extra usadas con <a href="#">CREATE TABLE</a> .
Comment	El comentario usado cuando se creó la tabla (o cierta información sobre el motivo por el que MySQL no puede acceder a la información de la tabla).

En el comentario de la tabla, para tabla **InnoDB** se informará sobre el espacio libre del espacio de tabla al que pertenece la tabla. Para una tabla localizada en el espacio de tablas compartidas, es el espacio libre para el espacio de tablas compartidas. Si se están usando varios espacios de tablas y la tabla tiene su propio espacio de tabla, el espacio libre es sólo para esa tabla.

Para tablas **MEMORY (HEAP)**, los valores de `Data_length`, `Max_data_length` e `Index_length` son aproximadamente la cantidad de memoria reservada actualmente. El algoritmo de reserva de memoria obtiene bloques de gran tamaño para reducir el número de operaciones de obtención de memoria.

Para vistas, todos los campos mostrados por **SHOW TABLE STATUS** son *NULL* excepto para 'Name' que indica el nombre de la vista y 'Comment' que dice 'view'.

(4.1.1)

# SHOW TABLES

```
SHOW [FULL|OPEN] TABLES [FROM nombre_db] [LIKE 'patrón']
```

**SHOW TABLES** lista las tablas no temporales en una base de datos dada. También se puede obtener esta lista usando el comando *mysqlshow db\_name*.

Antes de MySQL 5.0.1, la salida de **SHOW TABLES** contiene una única columna con los nombres de las tablas. A partir de MySQL 5.0.1, también se listan las vistas de la base de datos. A partir de MySQL 5.0.2, se soporta el modificador *FULL* de modo que **SHOW FULL TABLES** muestra una segunda columna. Los valores de esta segunda columna son *BASE TABLE* para una tabla y *VIEW* para una vista.

**Nota:** si no se dispone de privilegios para una tabla, la tabla no será mostrada en la salida de **SHOW TABLES** o *mysqlshow db\_name*.

**SHOW OPEN TABLES** lista las tablas que estén actualmente abiertas en la caché de tablas. El campo de comentario en la salida indica las veces que la tabla está en el caché y en uso. *OPEN* puede usarse a partir de MySQL 3.23.33.

(4.1.1)

# SHOW VARIABLES

```
SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'pattern']
```

**SHOW VARIABLES** muestra el valor de algunas variables de sistema de MySQL. Esta información puede ser obtenida también usando el comando *mysqladmin variables*.

Las opciones *GLOBAL* y *SESSION* son nuevas en MySQL 4.0.3. Con *GLOBAL*, se obtendrán los valores que se usarán para nuevas conexiones a MySQL. Con *SESSION*, los valores que están en uso para la conexión actual. Si no se usa ninguna, la opción por defecto es *SESSION*. *LOCAL* es sinónimo de *SESSION*.

Si los valores por defecto no son adecuados, se pueden asignar muchas de estas variables usando las opciones de línea de comandos cuando se arranca *mysqld* o durante la ejecución con la sentencia [SET](#).

A continuación se muestra una salida parcial. La lista de variables y sus valores pueden ser diferentes en cada servidor. El significado de cada variable se da en la sección 5.2.3 "Server System Variables" del manual de MySQL original. La información sobre su ajuste se proporciona en la sección 7.5.2 "Tuning Server Parameters".

```
mysql> SHOW VARIABLES;
```

Variable_name	Value
back_log	50
basedir	/usr/local/mysql
bdb_cache_size	8388572
bdb_log_buffer_size	32768
bdb_home	/usr/local/mysql
...	
max_connections	100
max_connect_errors	10
max_delayed_threads	20
max_error_count	64
max_heap_table_size	16777216
max_join_size	4294967295
max_relay_log_size	0
max_sort_length	1024
...	
timezone	EEST
tmp_table_size	33554432

## SHOW VARIABLES

```
| tmpdir | /tmp/:/mnt/hd2/tmp/ |  
| version | 4.0.4-beta |  
| wait_timeout | 28800 |  
+-----+-----+
```

Con la cláusula *LIKE*, la sentencia muestra sólo aquellas variables que coincidan con el patrón:

```
mysql> SHOW VARIABLES LIKE 'have%';  
+-----+-----+  
| Variable_name | Value |  
+-----+-----+  
| have_bdb | YES |  
| have_innodb | YES |  
| have_isam | YES |  
| have_raid | NO |  
| have_symlink | DISABLED |  
| have_openssl | YES |  
| have_query_cache | YES |  
+-----+-----+
```

(4.1.1)

# SHOW WARNINGS

```
SHOW WARNINGS [LIMIT [offset,] row_count]
SHOW COUNT(*) WARNINGS
```

**SHOW WARNINGS** muestra los mensajes de error, aviso y notas que resulten de la última sentencia que haya generado mensajes, o nada si la última sentencia que ha usado la tabla no ha generado mensajes. Esta sentencia se implementa a partir de MySQL 4.1.0. Una sentencia relacionada, [SHOW ERRORS](#), muestra sólo los errores.

La lista de mensajes se elimina para cada nueva sentencia que use una tabla.

La sentencia **SHOW COUNT(\*) WARNINGS** muestra el número total de errores, avisos y notas. También se puede recuperar este número a partir de la variable *warning\_count*:

```
SHOW COUNT(*) WARNINGS;
SELECT @@warning_count;
```

El valor de *warning\_count* puede ser mayor que el número de mensajes mostrados por **SHOW WARNINGS** si la variable de sistema *max\_error\_count* está asignada a un valor menor que hace que no todos los mensajes sean almacenados. Un ejemplo mostrado más abajo demuestra como puede ocurrir esto.

La cláusula *LIMIT* tiene la misma sintaxis que para la sentencia [SELECT](#).

El servidor MySQL devuelve el número total de errores, avisos y notas que resulten de la última sentencia. Si se usa el API C, este valor se puede obtener mediante una llamada a la función [mysql\\_warning\\_count\(\)](#).

Hay que tener en cuenta que el marco de trabajo para avisos se añadió en MySQL 4.1.0, en ese punto muchas sentencias no generan avisos. En 4.1.1, la situación es mucho mejor, con generación de avisos para sentencias como para [LOAD DATA INFILE](#) y sentencias **DML** como [INSERT](#), [UPDATE](#), [CREATE TABLE](#) y [ALTER TABLE](#).

La siguiente sentencia [DROP TABLE](#) generará una nota:

```
mysql> DROP TABLE IF EXISTS no_such_table;
mysql> SHOW WARNINGS;
+-----+-----+-----+
| Level | Code | Message |
+-----+-----+-----+
| Note  | 1051 | Unknown table 'no_such_table' |
+-----+-----+-----+
```

Sigue un ejemplo sencillo que muestra un aviso de sintaxis para [CREATE TABLE](#) y avisos de conversión para [INSERT](#):

```
mysql> CREATE TABLE t1 (a TINYINT NOT NULL, b CHAR(4)) TYPE=MyISAM;
Query OK, 0 rows affected, 1 warning (0.00 sec)
mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Warning
Code: 1287
Message: 'TYPE=storage_engine' is deprecated, use
        'ENGINE=storage_engine' instead
1 row in set (0.00 sec)

mysql> INSERT INTO t1 VALUES(10,'mysql'),(NULL,'test'),
-> (300,'Open Source');
Query OK, 3 rows affected, 4 warnings (0.01 sec)
Records: 3 Duplicates: 0 Warnings: 4

mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Warning
Code: 1265
Message: Data truncated for column 'b' at row 1
***** 2. row *****
Level: Warning
Code: 1263
Message: Data truncated, NULL supplied to NOT NULL column 'a' at row 2
***** 3. row *****
Level: Warning
Code: 1264
Message: Data truncated, out of range for column 'a' at row 3
***** 4. row *****
Level: Warning
Code: 1265
Message: Data truncated for column 'b' at row 3
4 rows in set (0.00 sec)
```

El número máximo de mensajes de error, aviso y notas a almacenar se controlan mediante la variable de sistema `max_error_count`. Por defecto, su valor es 64. Para cambiar el número de mensajes que se quieren almacenar, modificar el valor de `max_error_count`. En el siguiente ejemplo, la sentencia `ALTER TABLE` produce tres avisos, pero sólo uno se almacena porque el valor de `max_error_count` se ha puesto a 1:

```
mysql> SHOW VARIABLES LIKE 'max_error_count';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| max_error_count | 64    |
+-----+-----+
1 row in set (0.00 sec)

mysql> SET max_error_count=1;
Query OK, 0 rows affected (0.00 sec)

mysql> ALTER TABLE t1 MODIFY b CHAR;
Query OK, 3 rows affected, 3 warnings (0.00 sec)
Records: 3 Duplicates: 0 Warnings: 3

mysql> SELECT @@warning_count;
+-----+
| @@warning_count |
+-----+
|                3 |
+-----+
1 row in set (0.01 sec)

mysql> SHOW WARNINGS;
+-----+-----+-----+
| Level   | Code | Message                                     |
+-----+-----+-----+
| Warning | 1263 | Data truncated for column 'b' at row 1 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

Para desactivar los avisos, asignar 0 a `max_error_count`. En ese caso, `warning_count` sigue indicando cuantos avisos se han producido, pero no se almacena ningún mensaje.

#### (4.1.1)

# TRUNCATE

```
TRUNCATE TABLE table_name
```

**TRUNCATE TABLE** vacía una tabla por completo. Lógicamente, esto es equivalente a una sentencia **DELETE** que borre todas las filas, pero existen diferencias prácticas bajo algunas circunstancias.

Para **InnoDB**, **TRUNCATE TABLE** es mapeado a **DELETE**, de modo que no hay diferencia. para otros motores de almacenamiento, **TRUNCATE TABLE** difiere de **DELETE FROM ...** en lo siguiente, desde MySQL 4.0:

- El truncado trabaja suprimiendo y recreando la tabla, que es mucho más rápido que borrar filas una a una.
- **TRUNCATE** no es seguro a nivel de transacción; se puede obtener un error si se tienen transacciones activas o un bloqueo de tabla activo.
- No se devuelve el número de filas eliminadas.
- Mientras el fichero de definición de tabla 'table\_name.frm' sea válido, la tabla puede ser recreada como una tabla vacía con **TRUNCATE TABLE**, aunque los ficheros de datos o índices estén corruptos.
- El manipulador de tabla no recordará el último valor usado para **AUTO\_INCREMENT**, pero empezará a contar desde el principio. Esto es cierto incluso para tablas **MyISAM**, que generalmente no reutiliza los valores de secuencia.

En la versión 3.23 **TRUNCATE TABLE** se ejecuta como **COMMIT; DELETE FROM table\_name**, de modo que se comporta como **DELETE**.

**TRUNCATE TABLE** es una extensión SQL de Oracle. Esta sentencia fue añadida a MySQL 3.23.28, aunque desde 3.23.28 a 3.23.32, la palabra clave **TABLE** debe ser omitida.

(4.1.1)

# UNION

```
SELECT ...
UNION [ALL | DISTINCT]
SELECT ...
  [UNION [ALL | DISTINCT]
   SELECT ...]
```

**UNION** se usa para combinar los resultados de varias sentencias [SELECT](#) en un único conjunto de resultados. **UNION** está disponible a partir de MySQL 4.0.0.

Las columnas seleccionadas listadas en las posiciones correspondientes para cada sentencia [SELECT](#) deben ser del mismo tipo. (Por ejemplo, la primera columna seleccionada por la primera sentencia debe ser del mismo tipo que la primera columna seleccionada por las otras sentencias.) En los resultados devueltos se usarán los nombres de columna usados en la primera sentencia [SELECT](#).

Las sentencias [SELECT](#) son sentencias de selección normales, pero con las siguientes restricciones:

- Sólo la última sentencia [SELECT](#) puede tener una cláusula *INTO OUTFILE*.
- No se puede usar *HIGH\_PRIORITY* con las sentencias [SELECT](#) que formen parte de una **UNION**. Si se especifica para el primer [SELECT](#), no tendrá efecto. Si se hace para cualquier sentencia [SELECT](#) subsiguiente, se producirá un error de sintaxis.

Si no se usa la palabra clave *ALL* para la **UNION**, todas las filas devueltas serán únicas, igual que si se hubiese especificado *DISTINCT* para el conjunto de resultados total. Si se especifica *ALL*, se obtendrán todas las filas coincidentes de todas las sentencias [SELECT](#).

La palabra clave *DISTINCT* es una palabra opcional (introducida en MySQL 4.0.17). No hace nada, pero se admite en la sintaxis tal como se requiere en SQL estándar.

Antes de MySQL 4.1.2, no era posible mezclar **UNION ALL** y **UNION DISTINCT** en la misma consulta. Si se usa *ALL* para una **UNION**, se usará para todas ellas. A partir de MySQL 4.1.2, los tipos de **UNION** mezclados se tratan como si una unión *DISTINCT* anulara cualquier unión *ALL* a su izquierda. Una unión *DISTINCT* se puede producir explícitamente usando **UNION DISTINCT** o sencillamente **UNION** sin las palabras *DISTINCT* o *ALL* a continuación.

Si se quiere usar una cláusula *ORDER BY* o *LIMIT* para ordenar o limitar el resultado completo de una **UNION**, hay que poner entre paréntesis cada sentencia [SELECT](#) individual y colocar el *ORDER BY* o

*LIMIT* después de la última. El ejemplo siguiente usa ambas cláusulas:

```
(SELECT a FROM tbl_name WHERE a=10 AND B=1)
UNION
(SELECT a FROM tbl_name WHERE a=11 AND B=2)
ORDER BY a LIMIT 10;
```

Este tipo de *ORDER BY* no puede usar referencias de columna que incluyan un nombre de tabla (esto es, nombres en el formato `tbl_name.col_name`). En su lugar, hay que proporcionar un alias de columna en la primera sentencia [SELECT](#) y referirse al alias en la cláusula *ORDER BY*, o si no referirse a la columna en el *ORDER BY* usando su posición. (Es preferible un alias ya que el uso de posiciones de columna está desaconsejado.)

Para aplicar *ORDER BY* o *LIMIT* a un [SELECT](#) individual, hay que colocar la cláusula dentro del paréntesis que contiene el [SELECT](#):

```
(SELECT a FROM tbl_name WHERE a=10 AND B=1 ORDER BY a LIMIT 10)
UNION
(SELECT a FROM tbl_name WHERE a=11 AND B=2 ORDER BY a LIMIT 10);
```

Los tipos y longitudes de las columnas en el conjunto de resultados de una **UNION** tienen en cuenta los valores recuperados por todas las sentencias [SELECT](#). Antes de MySQL 4.1.1, una limitación de **UNION** es que sólo los valores del primer [SELECT](#) se usan para determinar los tipos y longitudes de las columnas del resultado. Esto puede producir valores truncados, por ejemplo, el primer [SELECT](#) devuelve valores más cortos que el segundo:

```
mysql> SELECT REPEAT('a',1) UNION SELECT REPEAT('b',10);
+-----+
| REPEAT('a',1) |
+-----+
| a              |
| b              |
+-----+
```

Esta limitación se ha eliminado desde MySQL 4.1.1:

```
mysql> SELECT REPEAT('a',1) UNION SELECT REPEAT('b',10);
+-----+
```

## UNION

```
| REPEAT('a',1) |  
+-----+  
| a             |  
| bbbbbbbbbbb |  
+-----+
```

(4.1.1)

# UPDATE

```
UPDATE [LOW_PRIORITY] [IGNORE] tbl_name
      SET col_name1=expr1 [, col_name2=expr2 ...]
      [WHERE where_definition]
      [ORDER BY ...]
      [LIMIT row_count]
```

O:

```
UPDATE [LOW_PRIORITY] [IGNORE] tbl_name [, tbl_name ...]
      SET col_name1=expr1 [, col_name2=expr2 ...]
      [WHERE where_definition]
```

**UPDATE** actualiza columnas de filas existentes de una tabla con nuevos valores. La cláusula *SET* indica las columnas a modificar y los valores que deben tomar. La cláusula *WHERE*, si se da, especifica qué filas deben ser actualizadas. Si no se especifica, serán actualizadas todas ellas. Si se especifica la cláusula *ORDER BY*, las filas se modificarán en el orden especificado. La cláusula *LIMIT* establece un límite al número de filas que se pueden actualizar.

La sentencia **UPDATE** soporta los modificadores siguientes:

- Si se usa la palabra *LOW\_PRIORITY*, la ejecución de **UPDATE** se retrasará hasta que no haya otros clientes haciendo lecturas de la tabla.
- Si se especifica *IGNORE*, la sentencia **UPDATE** no se abortará si se producen errores durante la actualización. Las filas con conflictos de claves duplicadas no se actualizarán. Las filas para las que la actualización de columnas se puedan producir errores de conversión se actualizarán con los valores válidos más próximos.

Si se accede a una columna de "tbl\_name" en una expresión, **UPDATE** usa el valor actual de la columna. Por ejemplo, la siguiente sentencia asigna a la columna "edad" su valor actual más uno:

```
mysql> UPDATE persondata SET edad=edad+1;
```

Las asignaciones **UPDATE** se evalúan de izquierda a derecha. Por ejemplo, las siguientes sentencias doblan el valor de la columna "edad", y después la incrementan:

```
mysql> UPDATE persondata SET edad=edad*2, edad=edad+1;
```

Si se asigna a una columna el valor que tiene actualmente, MySQL lo notifica y no la actualiza.

Si se actualiza una columna que ha sido declarada como *NOT NULL* con el valor *NULL*, se asigna el valor por defecto apropiado para el tipo de la columna y se incrementa en contador de avisos. El valor por defecto es 0 para tipos numéricos, la cadena vacía (") para tipos de cadena, y el valor "cero" para tipos de fecha y tiempo.

**UPDATE** devuelve el número de filas que se han modificado. A partir de la versión 3.22 de MySQL, la función de API C [mysql\\_info](#) devuelve el número de filas que han coincidido y actualizado, y el número de avisos que se han obtenido durante la actualización.

Desde la versión 3.23 de MySQL, se puede usar *LIMIT row\_count* para restringir el rango de actualización. La cláusula *LIMIT* trabaja del modo siguiente:

- Antes de MySQL 4.0.13, *LIMIT* restringía el número de filas afectadas. La sentencia se detiene tan pronto como se modifican "row\_count" filas que satisfagan la cláusula *WHERE*.
- Desde 4.0.13, *LIMIT* se restringe al número de filas coincidentes. La sentencia se detiene tan pronto como se encuentran "row\_count" filas que satisfagan la cláusula *WHERE*, tanto si se han modificado como si no.

Si se usa una cláusula *ORDER BY*, las filas serán actualizadas en el orden especificado. *ORDER BY* está disponible desde MySQL 4.0.0.

Desde la versión 4.0.4 de MySQL, también es posible realizar operaciones **UPDATE** que cubran múltiples tablas:

```
UPDATE items,month SET items.price=month.price
WHERE items.id=month.id;
```

El ejemplo muestra una fusión interna usando el operador coma, pero un **UPDATE** multitabla puede usar cualquier tipo de fusión (join) permitido en sentencias [SELECT](#), como un *LEFT JOIN*.

Nota: no es posible usar *ORDER BY* o *LIMIT* con **UPDATE** multitabla.

Si se usa una sentencia **UPDATE** multitabla que afecte a tablas **InnoDB** para las que haya definiciones de claves foráneas, el optimizador MySQL procesará las tablas en un orden diferente del de la relación padre/hijo. En ese caso, la sentencia puede fallar y deshará los cambios (roll back). En su lugar, se debe

actualizar una tabla y confiar en las capacidades de *ON UPDATE* que proporciona **InnoDB** que harán que las otras tablas se modifiquen del modo adecuado.

Actualmente, no se puede actualizar una tabla y seleccionar desde la misma en una subconsulta.

(4.1.1)

# USE

```
USE db_name
```

La sentencia **USE db\_name** indica a MySQL que use la base de datos *db\_name* como la base de datos por defecto (actual) en sentencias subsiguientes. La base de datos sigue siendo la base de datos por defecto hasta el final de la sesión o hasta que se use otra sentencia **USE**:

```
mysql> USE db1;
mysql> SELECT COUNT(*) FROM mytable;    # selecciona desde db1.mytable
mysql> USE db2;
mysql> SELECT COUNT(*) FROM mytable;    # selecciona desde db2.mytable
```

Hacer que una base de datos determinada sea la actual mediante el uso de la sentencia **USE** no descarta que se pueda acceder a tablas de otras bases de datos. El ejemplo siguiente accede a la tabla *author* de la base de datos *db1* y a la tabla *editor* de la base de datos *db2*:

```
mysql> USE db1;
mysql> SELECT author_name,editor_name FROM author,db2.editor
->      WHERE author.editor_id = db2.editor.editor_id;
```

La sentencia **USE** se proporciona por compatibilidad con **Sybase**.

(4.1.1)

# Indice de Funciones SQL (194)

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

**-A-** 

[ABS](#)

[ADDDATE](#)

[AES\\_DECRYPT](#)

[ASCII](#)

[ATAN](#)

[AVG](#)

[ACOS](#)

[ADDTIME](#)

[AES\\_ENCRYPT](#)

[ASIN](#)

[ATAN2](#)

**-B-** 

[BENCHMARK](#)

[BIT\\_AND](#)

[BIT\\_OR](#)

[BIN](#)

[BIT\\_LENGTH](#)

[BIT\\_XOR](#)

**-C-** 

[CAST](#)

[CEILING](#)

[CHARACTER\\_LENGTH](#)

[CHAR\\_LENGTH](#)

[COLLATION](#)

[CONCAT](#)

[CONNECTION\\_ID](#)

[CONVERT](#)

[COS](#)

[COUNT](#)

[CRC32](#)

[CURRENT\\_DATE](#)

[CURRENT\\_TIMESTAMP](#)

[CURTIME](#)

[CEIL](#)

[CHAR](#)

[CHARSET](#)

[COERCIBILITY](#)

[COMPRESS](#)

[CONCAT\\_WS](#)

[CONV](#)

[CONVERT\\_TZ](#)

[COT](#)

[COUNT DISTINCT](#)

[CURDATE](#)

[CURRENT\\_TIME](#)

[CURRENT\\_USER](#)

**-D-** 

DATABASEDATEDIFFDATE\_FORMATDAYDAYOFMONTHDAYOFYEARDEFAULTDES\_DECRYPTDIVDATEDATE\_ADDDATE\_SUBDAYNAMEDAYOFWEEKDECODEDEGREESDES\_ENCRYPT**-E-**ELTENCRYPTEXPORT\_SETENCODEEXPEXTRACT**-F-**FIELDFLOORFOUND\_ROWSFROM\_UNIXTIMEFIND\_IN\_SETFORMATFROM\_DAYS**-G-**GET\_FORMATGREATESTGET\_LOCKGROUP\_CONCAT**-H-**HEXHOUR**-I-**IFINET\_ATONINSERTIS\_FREE\_LOCKIFNULLINET\_NTOAINSTRIS\_USED\_LOCK**-L-**

LAST\_DAYLCASELEFTLNLOCALTIMELOCATELOG10LOWERLTRIMLAST\_INSERT\_IDLEASTLENGTHLOAD\_FILELOCALTIMESTAMPLOGLOG2LPAD**-M-** MAKEDATEMAKE\_SETMAXMICROSECONDMINMODMONTHNAMEMAKETIMEMASTER\_POS\_WAITMD5MIDMINUTEMONTH**-N-** NOWNULLIF**-O-** OCTOCTET\_LENGTHOLD\_PASSWORDORD**-P-** PASSWORDPERIOD\_ADDPERIOD\_DIFFPIPOSITIONPOWPOWER**-Q-** QUARTERQUOTE**-R-** 

RADIANSRELEASE\_LOCKREPLACERIGHTRPADRANDREPEATREVERSEROUNDRTRIM**-S-**SECONDSESSION\_USERSHA1SINSOUNDS\_LIKESQRTSTDDEVSTR\_TO\_DATESUBSTRINGSUBTIMESYSDATESEC\_TO\_TIMESHASIGNSOUNDEXSPACESTDSTRCMPSUBDATESUBSTRING\_INDEXSUMSYSTEM\_USER**-T-**TANTIMEDIFFTIMESTAMPADDTIME\_FORMATTO\_DAYSTRUNCATETIMETIMESTAMPTIMESTAMPDIFFTIME\_TO\_SECTRIM**-U-**UCASEUNCOMPRESSED\_LENGTHUNIX\_TIMESTAMPUSERUTC\_TIMEUNCOMPRESSUNHEXUPPERUTC\_DATEUTC\_TIMESTAMP

UUID

-V- 

VARIANCE

VERSION

-W- 

WEEK

WEEKDAY

WEEKOFYEAR

-Y- 

YEAR

YEARWEEK

# ABS

ABS(X)

Devuelve el valor absoluto de X:

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

mysql> SELECT ABS(-32);
+-----+
| ABS(-32) |
+-----+
|       32 |
+-----+
1 row in set (0.00 sec)
```

Esta función es segura con valores *BIGINT*.

# ACOS

ACOS(X)

Devuelve el arcocoseno de X, es decir, el valor del arco cuyo coseno es X. Devuelve NULL si X no está en el rango de -1 a 1:

```
mysql> SELECT ACOS(1);
+-----+
| ACOS(1) |
+-----+
| 0.000000 |
+-----+
1 row in set (0.02 sec)
```

```
mysql> SELECT ACOS(1.0001);
+-----+
| ACOS(1.0001) |
+-----+
|          NULL |
+-----+
1 row in set (0.02 sec)
```

```
mysql> SELECT ACOS(0);
+-----+
| ACOS(0) |
+-----+
| 1.570796 |
+-----+
1 row in set (0.00 sec)
```

# ADDDATE()

```
ADDDATE(date, INTERVAL expr type)
ADDDATE(expr, days)
```

Cuando se invoca con el formato *INTERVAL* para el segundo argumento, **ADDDATE()** es sinónimo de [DATE\\_ADD\(\)](#). La función relacionada [SUBDATE\(\)](#) es sinónimo de [DATE\\_SUB\(\)](#).

```
mysql> SELECT DATE_ADD('1998-01-02', INTERVAL 31 DAY);
+-----+
| DATE_ADD('1998-01-02', INTERVAL 31 DAY) |
+-----+
| 1998-02-02                               |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT ADDDATE('1998-01-02', INTERVAL 31 DAY);
+-----+
| ADDDATE('1998-01-02', INTERVAL 31 DAY) |
+-----+
| 1998-02-02                               |
+-----+
1 row in set (0.00 sec)
```

Desde MySQL 4.1.1, se permite la segunda sintaxis, donde *expr* es una fecha o una expresión 'datetime' y *days* es el número de días a añadir a *expr*.

```
mysql> SELECT ADDDATE('1998-01-02', 31);
+-----+
| ADDDATE('1998-01-02', 31) |
+-----+
| 1998-02-02                               |
+-----+
```

# ADDTIME()

```
ADDTIME ( expr , expr2 )
```

**ADDTIME()** añade `expr2` a `expr` y devuelve el resultado. `expr` es una fecha o una expresión de tipo `datetime`, y `expr2` es una expresión de tipo `time`.

```
mysql> SELECT ADDTIME("1997-12-31 23:59:59.999999", "1 1:1:1.000002");
      -> '1998-01-02 01:01:01.000001'
mysql> SELECT ADDTIME("01:00:00.999999", "02:00:00.999998");
      -> '03:00:01.999997'
```

**ADDTIME()** fue añadida en MySQL 4.1.1.

# AES\_ENCRYPT

# AES\_DECRYPT

```
AES_ENCRYPT(string,key_string)
AES_DECRYPT(string,key_string)
```

Estas funciones permiten encriptar y desencriptar datos usando el algoritmo oficial AES (Advanced Encryption Standard), conocido previamente como Rijndael. Se usa una codificación con una clave de 128 bits de longitud, pero se puede extender a 256 modificando el fuente. Se ha seleccionado 128 bits porque es mucho más rápido y normalmente es proporciona suficiente seguridad. Los argumentos de entrada pueden ser de cualquier longitud. Si cualquier argumento es NULL, el resultado de la función es también NULL. Como AES es un algoritmo a nivel de bloque, se usa relleno para para codificar cadenas de longitud irregular y entonces la longitud de la cadena resultante puede ser calculada como  $16 * (\text{trunc}(\text{string\_length}/16) + 1)$ . Si **AES\_DECRYPT()** detecta datos inválidos o un relleno incorrecto, devuelve NULL. Además, es posible que **AES\_DECRYPT()** devuelva un valor no NULL (seguramente basura) si los datos de entrada o la clave son inválidos. Se pueden usar la funciones *AES* para almacenar datos en un formato encriptado modificando las consultas:

```
INSERT INTO t VALUES (1,AES_ENCRYPT('text','password'));
```

Se puede obtener mayor seguridad no transfiriendo la clave sobre la conexión para cada consulta, esto puede conseguirse almacenandola en una variable del servidor durante la conexión:

```
SELECT @password:='my password';
INSERT INTO t VALUES (1,AES_ENCRYPT('text',@password));
```

**AES\_ENCRYPT()** y **AES\_DECRYPT()** fueron añadidas en la versión 4.0.2, y pueden considerarse como las funciones de encriptado criptográfico más seguras disponibles actualmente en MySQL.

# ASCII()

```
ASCII(str)
```

Devuelve el valor de código ASCII del carácter más a la izquierda de la cadena str. Devuelve 0 si str es una cadena vacía. Devuelve NULL si str es NULL:

```
mysql> SELECT ASCII('2');
```

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

```
| ASCII('2') |
```

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

```
|          50 |
```

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

```
1 row in set (0.03 sec)
```

```
mysql> SELECT ASCII(2);
```

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

```
| ASCII(2) |
```

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

```
|          50 |
```

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

```
1 row in set (0.00 sec)
```

```
mysql> SELECT ASCII('dx');
```

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

```
| ASCII('dx') |
```

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

```
|          100 |
```

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

```
1 row in set (0.00 sec)
```

Ver también la función [ORD\(\)](#).

# ASIN

ASIN(X)

Devuelve el arcoseno de X, es decir, el valor del arco cuyo seno es X. Devuelve NULL si X no está en el rango de -1 a 1:

```
mysql> SELECT ASIN(0.2);
```

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

```
| ASIN(0.2) |
```

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

```
| 0.201358 |
```

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

```
1 row in set (0.00 sec)
```

```
mysql> SELECT ASIN('foo');
```

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

```
| ASIN('foo') |
```

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

```
| 0.000000 |
```

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

```
1 row in set (0.01 sec)
```

# ATAN

## ATAN2

```
ATAN(X)
ATAN(Y,X)
ATAN2(Y,X)
```

Devuelve el arcotangente de X, es decir, el valor del arco cuya tangente es X:

```
mysql> SELECT ATAN(2);
+-----+
| ATAN(2) |
+-----+
| 1.107149 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT ATAN(-2);
+-----+
| ATAN(-2) |
+-----+
| -1.107149 |
+-----+
1 row in set (0.00 sec)
```

Con dos argumentos devuelve el arcotangente de las dos variables X e Y. Esto es similar a calcular el arcotangente de  $Y / X$ , excepto que los sigbos de ambos argumentos se tienen en cuenta para determinar el cuadrante del resultado:

```
mysql> SELECT ATAN(-2,2);
+-----+
| ATAN(-2,2) |
+-----+
| -0.785398 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT ATAN2(PI(),0);
+-----+
| ATAN2(PI(),0) |
```

```
+-----+  
|      1.570796 |  
+-----+  
1 row in set (0.00 sec)
```

# AVG

```
AVG(expr)
```

Devuelve el valor medio de expr:

```
mysql> SELECT student_name, AVG(test_score)
->      FROM student
->      GROUP BY student_name;
```

Si se usa una función de grupo en una sentencia que contenga la cláusula *GROUP BY*, equivale a agrupar todas las filas.

# BENCHMARK

```
BENCHMARK ( count , expr )
```

La función **BENCHMARK()** ejecuta la expresión `expr` `count` veces. Puede usarse para medir cuan rápido procesa la expresión MySQL. El valor resultado siempre es cero. La intención es usarla en el cliente `mysql`, que informa del tiempo que ha requerido la ejecución de la consulta:

```
mysql> SELECT BENCHMARK(1000000,ENCODE("hello","goodbye"));
+-----+
| BENCHMARK(1000000,ENCODE("hello","goodbye")) |
+-----+
|                                          0 |
+-----+
1 row in set (4.74 sec)
```

El tiempo mostrado es el transcurrido en la parte del cliente, no el tiempo de la CPU en el extremo del servidor. Es aconsejable ejecutar **BENCHMARK()** algunas veces, e interpretar el resultado en el sentido del nivel de carga que tiene la máquina del servidor.

# BIN()

`BIN(N)`

Devuelve una cadena que representa el valor binario de N, donde N es un número longlong (BIGINT). Es equivalente a [CONV\(N,10,2\)](#). Devuelve NULL si N es NULL:

```
mysql> SELECT BIN(12);
+-----+
| BIN(12) |
+-----+
| 1100    |
+-----+
1 row in set (0.01 sec)
```

# BIT\_AND

```
BIT_AND(expr)
```

Devuelve la operación de bits AND para todos los bits de expr. El cálculo se realiza con precisión de 64 bits (*BIGINT*). Desde MySQL 4.0.17, esta función devuelve 18446744073709551615 si no existen filas que coincidan. (Es el valor *BIGINT* sin signo con todos los bits a 1.) Antes de 4.0.17, la función devolvía -1 en ese caso.

Si se usa una función de grupo en una sentencia que contenga la cláusula *GROUP BY*, equivale a agrupar todas las filas.

# BIT\_LENGTH()

```
BIT_LENGTH(str)
```

Devuelve la longitud de la cadena str en bits:

```
mysql> SELECT BIT_LENGTH('text');
+-----+
| BIT_LENGTH('text') |
+-----+
| 32                  |
+-----+
1 row in set (0.01 sec)
```

# BIT\_OR

```
BIT_OR ( expr )
```

Devuelve la operación de bits OR para todos los bits de expr. El cálculo se realiza con precisión de 64 bits (*BIGINT*). Esta función devuelve 0 si no existen filas que coincidan.

Si se usa una función de grupo en una sentencia que contenga la cláusula *GROUP BY*, equivale a agrupar todas las filas.

# BIT\_XOR

```
BIT_XOR(expr)
```

Devuelve la operación de bits XOR para todos los bits de expr. El cálculo se realiza con precisión de 64 bits (*BIGINT*). Esta función devuelve 0 si no existen filas que coincidan. Esta función está disponible desde MySQL 4.1.1.

Si se usa una función de grupo en una sentencia que contenga la cláusula *GROUP BY*, equivale a agrupar todas las filas.

# CAST CONVERT

```
CAST(expression AS type)
CONVERT(expression,type)
CONVERT(expr USING transcoding_name)
```

Las funciones **CAST()** y **CONVERT()** pueden usarse para tomar un valor de un tipo y obtener uno de otro tipo.

Los valores de 'type' pueden ser uno de los siguientes:

- BINARY
- CHAR
- DATE
- DATETIME
- SIGNED {INTEGER}
- TIME
- UNSIGNED {INTEGER}

**CAST()** y **CONVERT()** están disponibles desde MySQL 4.0.2. La conversión del tipo *CHAR* desde 4.0.6. La forma *USING* de **CONVERT()** está disponible desde 4.1.0.

**CAST()** y **CONVERT(... USING ...)** forman parte de la sintaxis de SQL-99. La forma de **CONVERT** sin *USING* pertenece a la sintaxis de **ODBC**.

Las funciones de conversión de tipo son corrientes cuando se quiere crear una columna de un tipo específico en una sentencia **CREATE ... SELECT**:

```
CREATE TABLE new_table SELECT CAST('2000-01-01' AS DATE);
```

También son útiles para ordenar columnas *ENUM* por orden alfabético. Normalmente, ordenar columnas *ENUM* usa los valores del orden numérico interno. Haciendo la conversión a *CHAR* resulta un orden alfabético:

```
SELECT enum_col FROM tbl_name ORDER BY CAST(enum_col AS CHAR);
```

**CAST(string AS BINARY)** es lo mismo que una cadena *BINARY*. **CAST(expr AS CHAR)** trata la expresión como una cadena con el juego de caracteres por defecto.

NOTA: En MySQL 4.0 la función **CAST()** para *DATE*, *DATETIME* o *TIME* sólo marca la columna para que sea de un tipo específico pero no cambia el valor de la columna.

En MySQL 4.1.0 el valor se convierte al tipo de columna correcto cuando se envía al usuario (esta es una característica del modo que que el nuevo protocolo en la versión 4.1 envía la información de fecha al cliente):

```
mysql> SELECT CAST(NOW() AS DATE);
-> 2003-05-26
```

En versiones más recientes de MySQL (probablemente en 4.1.2 o 5.0) se corregirá este **CAST** para que también cambie el resultado si se usa como parte de una expresión más compleja, como [\*CONCAT\*](#) ("*Date:* ",*CAST(NOW() AS DATE)*).

No se puede usar **CAST()** para extraer datos en diferentes formatos, pero en su lugar se pueden usar funciones de cadena como [\*LEFT\(\)\*](#) o [\*EXTRACT\(\)\*](#).

Para convertir una cadena a valor numérico, normalmente no hay que hacer nada; sencillamente se usa el valor de la cadena como si se tratase de un número:

```
mysql> SELECT 1+'1';
+-----+
| 1+'1' |
+-----+
|      2 |
+-----+
1 row in set (0.03 sec)
```

Si se usa un número en un contexto de cadena, el número se convertirá automáticamente a cadena *BINARY*:

```
mysql> SELECT CONCAT("hello you ",2);
+-----+
| CONCAT("hello you ",2) |
+-----+
```

```
| hello you 2 |
+-----+
1 row in set (0.02 sec)
```

MySQL soporta aritmética con valores de 64 bits, tanto con como sin signo. Si se usan operaciones numéricas (como +) y uno de los operandos es un entero sin signo, el resultado será sin signo. Se puede evitar esto usando una conversión de tipo *SIGNED* y *UNSIGNED* para los operadores para convertir el resultado a un entero de 64 bits con o sin signo, respectivamente.

```
mysql> SELECT CAST(1-2 AS UNSIGNED)
      -> 18446744073709551615
mysql> SELECT CAST(CAST(1-2 AS UNSIGNED) AS SIGNED);
      -> -1
```

Por otra parte, si cualquiera de los operandos es un valor en punto flotante, el resultado será un valor en punto flotante y no resulta afectado por la regla anterior. (En ese contexto, los valores *DECIMAL* serán promocionados a valores en punto flotante.)

```
mysql> SELECT CAST(1 AS UNSIGNED) - 2.0;
      -> -1.0
```

Si se usa una cadena en una operación aritmética, será convertida a un número en punto flotante.

La manipulación de valores con signo se cambió en MySQL 4.0 para permitir soportar valores *BIGINT* apropiadamente. Si se posee algún código que se pretende ejecutar tanto en MySQL 4.0 como en 3.23 (en cuyo caso probablemente no se podrá usar la función **CAST()**), se puede usar la siguiente técnica para obtener un resultado con signo cuando se restan dos columnas con enteros sin signo:

```
SELECT (unsigned_column_1+0.0)-(unsigned_column_2+0.0);
```

La idea es que las columnas se conviertan a valores en punto flotante antes de hacer la resta.

Si se tienen problemas con columnas *UNSIGNED* en una aplicación MySQL antigua cuando se porta a MySQL 4.0, se puede usar la opción `--sql-mode=NO_UNSIGNED_SUBTRACTION` cuando se arranca **mysqld**. Sin embargo, tan pronto se use esto, no será posible hacer un uso eficiente de columnas de tipo *BIGINT UNSIGNED*.

**CONVERT()** con *USING* se usa para convertir datos entre diferentes juegos de caracteres. En MySQL,

los nombres de traducción son los mismos que los nombres de los juegos de caracteres correspondientes. Por ejemplo, esta sentencia convierte la cadena 'abc' en el juego de caracteres por defecto del servidor a la cadena correspondiente en el juego de caracteres utf8:

```
SELECT CONVERT('abc' USING utf8);
```

# CEILING

## CEIL

```
CEILING(X)  
CEIL(X)
```

Devuelve el entero más pequeño cuyo valor es mayor que X:

```
mysql> SELECT CEILING(1.23);  
+-----+  
| CEILING(1.23) |  
+-----+  
|           2 |  
+-----+  
1 row in set (0.02 sec)
```

```
mysql> SELECT CEIL(-1.23);  
+-----+  
| CEIL(-1.23) |  
+-----+  
|          -1 |  
+-----+  
1 row in set (0.00 sec)
```

El alias **CEIL()** fue añadido en la versión 4.0.6. El valor de retorno se convierte a *BIGINT!*.

# CHAR()

```
CHAR(N, ...)
```

**CHAR()** interpreta los argumentos como enteros y devuelve una cadena que consiste en los caracteres dados por los valores de los códigos ASCII de esos enteros. Los valores NULL se saltan:

```
mysql> SELECT CHAR(77,121,83,81,'76');
+-----+
| CHAR(77,121,83,81,'76') |
+-----+
| MySQL                    |
+-----+
1 row in set (0.03 sec)
mysql> SELECT CHAR(77,77.3,'77.3');
+-----+
| CHAR(77,77.3,'77.3') |
+-----+
| MMM                   |
+-----+
1 row in set (0.02 sec)
```

# ***CHAR\_LENGTH()***

## ***CHARACTER\_LENGTH()***

```
CHAR_LENGTH(str)
CHARACTER_LENGTH(str)
```

Devuelve la longitud de la cadena `str`, medida en caracteres. Un carácter multibyte cuenta como un carácter sencillo. Esto significa que para una cadena que contenga cinco caracteres de dos bytes, [LENGTH\(\)](#) devuelve 10, mientras que **CHAR\_LENGTH()** devuelve 5.

**CHARACTER\_LENGTH()** es un sinónimo de **CHAR\_LENGTH()**.

# CHARSET

```
CHARSET(str)
```

Devuelve el conjunto de caracteres de la cadena argumento.

```
SELECT CHARSET('abc');  
      -> 'latin1'  
mysql> SELECT CHARSET(CONVERT('abc' USING utf8));  
      -> 'utf8'  
mysql> SELECT CHARSET(USER());  
      -> 'utf8'
```

**CHARSET()** se añadió en MySQL 4.1.0.

# COERCIBILITY

```
COERCIBILITY(str)
```

Devuelve el valor de restricción de colección de la cadena argumento.

```
mysql> SELECT COERCIBILITY('abc' COLLATE latin1_swedish_ci);  
      -> 0  
mysql> SELECT COERCIBILITY('abc');  
      -> 3  
mysql> SELECT COERCIBILITY(USER());  
      -> 2
```

Los valores de retorno tienen los siguientes significados:

Valor	Significado
0	Colección explícita
1	Sin colección
2	Colección implícita
3	Coercitivo

Los valores más bajos tienen la mayor precedencia. **COERCIBILITY()** se añadió en MySQL 4.1.1.

# COLLATION

```
COLLATION(str)
```

Devuelve la colección para el conjunto de caracteres de la cadena argumento.

```
mysql> SELECT COLLATION('abc');  
      -> 'latin1_swedish_ci'  
mysql> SELECT COLLATION(_utf8'abc');  
      -> 'utf8_general_ci'
```

**COLLATION()** se añadió en MySQL 4.1.0.

# COMPRESS

```
COMPRESS(string_to_compress)
```

Comprime una cadena.

```
mysql> SELECT LENGTH(COMPRESS(REPEAT("a",1000)));  
      -> 21  
mysql> SELECT LENGTH(COMPRESS(""));  
      -> 0  
mysql> SELECT LENGTH(COMPRESS("a"));  
      -> 13  
mysql> SELECT LENGTH(COMPRESS(REPEAT("a",16)));  
      -> 15
```

**COMPRESS()** fue añadido en MySQL 4.1.1. Requiere que MySQL se haya compilado con una librería de compresión como **zlib**. De otro modo, el valor de retorno es siempre NULL. El contenido de la cadena comprimida se almacena del siguiente modo:

- Las cadenas vacías se almacenan como cadenas vacías.
- Las cadenas no vacías se almacenan como un dato de 4 bytes con la longitud de la cadena sin comprimir (el byte de menor peso primero), seguido por la cadena comprimida usando gzip. Si la cadena termina con un espacio, se añade un '.' extra para impedir problemas con la posible eliminación de espacios si el resultado se almacena en una columna *CHAR* o *VARCHAR*. El uso de *CHAR* o *VARCHAR* para almacenar cadenas comprimidas no se recomienda. Es mucho mejor usar una columna *BLOB* en su lugar.

# CONCAT()

```
CONCAT(str1,str2,...)
```

Devuelve la cadena resultante de concatenar los argumentos. Devuelve NULL si alguno de los argumentos es NULL. Puede haber más de 2 argumentos. Un argumento numérico se convierte a su cadena equivalente:

```
mysql> SELECT CONCAT('My', 'S', 'QL');
```

```
+-----+
| CONCAT('My', 'S', 'QL') |
+-----+
| MySQL                    |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT CONCAT('My', NULL, 'QL');
```

```
+-----+
| CONCAT('My', NULL, 'QL') |
+-----+
| NULL                      |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT CONCAT(14.3);
```

```
+-----+
| CONCAT(14.3) |
+-----+
| 14.3         |
+-----+
1 row in set (0.00 sec)
```

# CONCAT\_WS()

```
CONCAT_WS(separator, str1, str2,...)
```

**CONCAT\_WS()** funciona como **CONCAT()** pero con separadores y es una forma especial de **CONCAT()**. El primer argumento es el separador para el resto de los argumentos. El separador se añade entre las cadenas a concatenar: El separador puede ser una cadena, igual que el resto de los argumentos. Si el separador es NULL, el resultado es NULL. La función pasa por alto cualquier valor NULL después del argumento separador.

```
mysql> SELECT CONCAT_WS(",","First name","Second name","Last Name");
+-----+
| CONCAT_WS(",","First name","Second name","Last Name") |
+-----+
| First name,Second name,Last Name |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT CONCAT_WS(",","First name",NULL,"Last Name");
+-----+
| CONCAT_WS(",","First name",NULL,"Last Name") |
+-----+
| First name,Last Name |
+-----+
1 row in set (0.00 sec)
```

Antes de MySQL 4.0.14, **CONCAT\_WS()** pasaba por alto las cadenas vacías del mismo modo que los valores NULL.

# CONNECTION\_ID

```
CONNECTION_ID()
```

Devuelve el ID (ID del hilo) de una conexión. Cada conexión tiene su propio y único ID:

```
mysql> SELECT CONNECTION_ID();
+-----+
| CONNECTION_ID() |
+-----+
|           188 |
+-----+
1 row in set (0.00 sec)
```

# CONV()

```
CONV(N, from_base, to_base)
```

Convierte números entre distintas bases. Devuelve una cadena que representa el número N, convertido desde la base `from_base` a la base `to_base`. Devuelve *NULL* si alguno de los argumentos es *NULL*. El argumento N se interpreta como un entero, pero puede ser especificado como un entero o como una cadena. La base mínima es 2 y la máxima 36. If `to_base` es un número negativo, N es tratado como un número con signo. En caso contrario, N se trata como sin signo. **CONV** trabaja con una precisión de 64 bits:

```
mysql> SELECT CONV("a",16,2);
+-----+
| CONV("a",16,2) |
+-----+
| 1010           |
+-----+
1 row in set (0.00 sec)

mysql> SELECT CONV("6E",18,8);
+-----+
| CONV("6E",18,8) |
+-----+
| 172             |
+-----+
1 row in set (0.00 sec)

mysql> SELECT CONV("-17",10,-18);
+-----+
| CONV("-17",10,-18) |
+-----+
| -H               |
+-----+
1 row in set (0.00 sec)

mysql> SELECT CONV(10+"10"+"10'+0xa",10,10);
+-----+
| CONV(10+"10"+"10'+0xa,10,10) |
+-----+
| 40                 |
+-----+
1 row in set (0.01 sec)
```



# CONVERT\_TZ

```
CONVERT_TZ(dt,from_tz,to_tz)
```

**CONVERT\_TZ()** convierte un valor `dt`, de fecha y hora, des la zona de tiempos `from_tz` dada a la zona de tiempos `to_tz` y devuelve el valor resultante. Las zonas de tiempo pueden ser especificadas como se describe en la sección [soporte de zonas de tiempo](#). Esta función devuelve *NULL* si el argumento es inválido. Si el valor cae fuera del rango soportado por el tipo *TIMESTAMP* cuando se convierte desde `from_tz` a UTC, no se realiza conversión alguna.

```
mysql> SELECT CONVERT_TZ('2004-01-01 12:00:00','GMT','MET');
      -> '2004-01-01 13:00:00'
mysql> SELECT CONVERT_TZ('2004-01-01 12:00:00','+00:00','-07:00');
      -> '2004-01-01 05:00:00'
```

Para usar zonas de tiempo con nombre como 'MET' o 'Europe/Moscow', la tabla de zonas de tiempo debe ser preparada convenientemente. **CONVERT\_TZ()** se añadió en MySQL 4.1.3.

# COS

```
COS(X)
```

Devuelve el coseno de X, donde X viene dado en radianes:

```
mysql> SELECT COS(PI());  
+-----+  
| COS(PI()) |  
+-----+  
| -1.000000 |  
+-----+  
1 row in set (0.00 sec)
```

# COT

COT(X)

Devuelve la cotangente de X:

```
mysql> SELECT COT(12);
+-----+
| COT(12) |
+-----+
| -1.57267341 |
+-----+
1 row in set (0.00 sec)
```

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

# COUNT

```
COUNT ( expr )
```

Devuelve un contador con el número de valores distintos de NULL en las filas recuperadas por una sentencia **SELECT**:

```
mysql> SELECT student.student_name, COUNT(*)
->      FROM student, course
->      WHERE student.student_id=course.student_id
->      GROUP BY student_name;
```

**COUNT(\*)** es algo diferente en que devuelve un contador con el número de filas recuperadas, contengan o no valores NULL. **COUNT(\*)** está optimizado para regresar mucho más rápido si la sentencia **SELECT** recupera de una tabla, no se piden otras columnas y no existe cláusula *WHERE*. Por ejemplo:

```
mysql> SELECT COUNT(*) FROM student;
```

Esta optimización se aplica sólo a tablas **MyISAM** y **ISAM**, ya que se almacena un registro de cuenta exacto para estos tipos de tabla y puede ser accedida muy rápidamente. Para máquinas de almacenamiento transaccionales (**InnoDB**, **BDB**), almacenar una fila de cuenta exacta es más problemático porque pueden ocurrir múltiples transacciones, y cada una puede afectar a la cuenta.

Si se usa una función de grupo en una sentencia que contenga la cláusula *GROUP BY*, equivale a agrupar todas las filas.

# COUNT DISTINCT

```
COUNT(DISTINCT expr,[expr...])
```

Devuelve un contador con el número de valores diferentes, distintos de NULL:

```
mysql> SELECT COUNT(DISTINCT results) FROM student;
```

En MySQL se puede obtener el número de una combinación de expresiones diferentes que no contengan NULL mediante una lista de expresiones. En SQL-99 se puede hacer una concatenación de todas las expresiones dentro de *COUNT(DISTINCT ...)*.

Si se usa una función de grupo en una sentencia que contenga la cláusula *GROUP BY*, equivale a agrupar todas las filas.

## CRC32

```
CRC32(expr)
```

Calcula el valor de comprobación de redundancia cíclica y devuelve un valor entero sin signo de 32 bits. El resultado es NULL si el argumento es NULL. El argumento esperado es una cadena y será tratado como tal si no lo es.

```
mysql> SELECT CRC32('MySQL');  
-> 3259397556
```

CRC32() is available as of MySQL 4.1.0.

# ***CURDATE()***

## ***CURRENT\_DATE***

## ***CURRENT\_DATE()***

**CURRENT\_DATE** y **CURRENT\_DATE()** son sinónimos de **CURDATE**.

```
CURDATE ( )
```

```
CURRENT_DATE
```

```
CURRENT_DATE ( )
```

Devuelve la fecha actual como un valor en el formato 'AAAA-MM-DD' o AAAAMMDD, dependiendo de si la función se usa en un contexto de cadena o numérico:

```
mysql> SELECT CURDATE();
+-----+
| CURDATE() |
+-----+
| 2003-12-16 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT CURDATE() + 0;
+-----+
| CURDATE() + 0 |
+-----+
|      20031216 |
+-----+
1 row in set (0.00 sec)
```

# ***CURTIME()***

## ***CURRENT\_TIME***

## ***CURRENT\_TIME()***

```
CURTIME()  
CURRENT_TIME  
CURRENT_TIME()
```

Devuelve la hora actual como un valor en el formato 'HH:MM:SS' o HHMMSS, dependiendo de si la función se usa en un contexto de cadena o numérico:

```
mysql> SELECT CURTIME();  
+-----+  
| CURTIME() |  
+-----+  
| 12:40:07 |  
+-----+  
1 row in set (0.00 sec)  
  
mysql> SELECT CURTIME() + 0;  
+-----+  
| CURTIME() + 0 |  
+-----+  
|          124015 |  
+-----+  
1 row in set (0.02 sec)
```

**CURRENT\_TIME** y **CURRENT\_TIME()** son sinónimos de **CURTIME()**.

**NOW()**  
**CURRENT\_TIMESTAMP**  
**CURRENT\_TIMESTAMP()**  
**LOCALTIME**  
**LOCALTIME()**  
**LOCALTIMESTAMP**  
**LOCALTIMESTAMP()**  
**SYSDATE()**

```
NOW( )
CURRENT_TIMESTAMP
CURRENT_TIMESTAMP( )
LOCALTIME
LOCALTIME( )
LOCALTIMESTAMP
LOCALTIMESTAMP( )
SYSDATE( )
```

Devuelve la fecha y hora actual como un valor en el formato 'YYYY-MM-DD HH:MM:SS' o YYYYMMDDHHMMSS, dependiendo de si la función se usa en un contexto de cadena o de número:

```
mysql> SELECT NOW();
+-----+
| NOW() |
+-----+
| 2003-12-26 21:04:10 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT NOW() + 0;
+-----+
| NOW() + 0 |
+-----+
| 20031226210432 |
+-----+
1 row in set (0.00 sec)
```

NOW

**CURRENT\_TIMESTAMP, CURRENT\_TIMESTAMP(), LOCALTIME, LOCALTIME(), LOCALTIMESTAMP, LOCALTIMESTAMP() y SYSDATE() son sinónimos de NOW().**

# CURRENT\_USER

```
CURRENT_USER( )
```

Devuelve el nombre de usuario y el del host para el que está autenticada la conexión actual. Este valor corresponde a la cuenta que se usa para evaluar los privilegios de acceso. Puede ser diferente del valor de [USER\(\)](#).

```
mysql> SELECT USER();
      -> 'davida@localhost'
mysql> SELECT * FROM mysql.user;
      -> ERROR 1044: Access denied for user: '@localhost' to database
'mysql'
mysql> SELECT CURRENT_USER();
      -> '@localhost'
```

El ejemplo ilustra que aunque el cliente ha especificado un nombre de usuario 'davida' (como indica el valor de la función [USER\(\)](#)), el servidor autenticó al cliente usando una cuenta de usuario anónima (como se ve por la parte del nombre de usuario vacía del valor **CURRENT\_USER()**). Un modo en el que puede ocurrir esto es que no exista una cuenta en las tablas de concesiones (grant) para 'davida'.

# ***DATABASE***

```
DATABASE( )
```

Devuelve el nombre de la base de datos actual:

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

Si no hay una base de datos actual, **DATABASE()** devuelve NULL en MySQL 4.1.1, y una cadena vacía antes de esa versión.

# DATE()

```
DATE ( expr )
```

Extrae la parte de la fecha de una expresión expre de tipo date o datetime.

```
mysql> SELECT DATE('2003-12-31 01:02:03');  
      -> '2003-12-31'
```

**DATE()** está disponible desde MySQL 4.1.1.

# DATEDIFF()

```
DATEDIFF( expr , expr2 )
```

**DATEDIFF()** devuelve el número de días entre la fecha de inicio `expr` y la de final `expr2`. `expr` y `expr2` son expresiones de tipo `date` o `date-and-time`. Sólo las partes correspondientes a la fecha de cada expresión se usan en los cálculos.

```
mysql> SELECT DATEDIFF('1997-12-31 23:59:59', '1997-12-30');  
      -> 1  
mysql> SELECT DATEDIFF('1997-11-31 23:59:59', '1997-12-31');  
      -> -30
```

**DATEDIFF()** está disponible desde MySQL 4.1.1.

# DATE\_ADD()

# DATE\_SUB()

```
DATE_ADD(date, INTERVAL expr type)
DATE_SUB(date, INTERVAL expr type)
```

Estas funciones realizan aritmética con fechas. Desde MySQL 3.23, *INTERVAL expr type* se permite en cualquiera de los lados del operador + si la expresión en el otro lado es un valor de tipo date o datetime. Para el operador -, *INTERVAL expr type* se permite sólo en el lado derecho, porque no tiene sentido restar un valor de tipo date o datetime desde un intervalo. (Ver ejemplos.) *date* es un valor de tipo *DATETIME* o *DATE* que especifica la fecha de comienzo. *expr* es una expresión que especifica un valor de intervalo a añadir o restar desde la fecha de comienzo. *expr* es una cadena; puede empezar con un '-' para intervalos negativos. *type* es una palabra clave que indica cómo debe interpretarse la expresión. La tabla siguiente muestra cómo se relacionan los argumentos *type* y *expr*:

Valor <i>type</i>	Formato de <i>expr</i> esperado
MICROSECOND	MICROSECONDS
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS
YEAR	YEARS
SECOND_MICROSECOND	'SECONDS.MICROSECONDS'
MINUTE_MICROSECOND	'MINUTES.MICROSECONDS'
MINUTE_SECOND	'MINUTES:SECONDS'
HOUR_MICROSECOND	'HOURS.MICROSECONDS'
HOUR_SECOND	'HOURS:MINUTES:SECONDS'
HOUR_MINUTE	'HOURS:MINUTES'
DAY_MICROSECOND	'DAYS.MICROSECONDS'
DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'

DAY_MINUTE	'DAYS HOURS:MINUTES'
DAY_HOUR	'DAYS HOURS'
YEAR_MONTH	'YEARS-MONTHS'

Los valores de tipo *DAY\_MICROSECOND*, *hour\_microsecond*, *MINUTE\_MICROSECOND*, *SECOND\_MICROSECOND*, and *MICROSECOND* están permitidos desde MySQL 4.1.1. Los valores *QUARTER* y *WEEK* están permitidos desde MySQL 5.0.0. MySQL permite cualquier delimitador de puntuación en el formato de expr. Los mostrados en la tabla son los delimitadores propuestos. Si el argumento date es un valor *DATE* y los cálculos involucran sólo las partes de *YEAR*, *MONTH* y *DAY* (es decir, no las partes de hora), el resultado es un valor *DATE*. En otro caso, el resultado es un valor *DATETIME*:

```
mysql> SELECT '1997-12-31 23:59:59' + INTERVAL 1 SECOND;
```

```
+-----+
| '1997-12-31 23:59:59' + INTERVAL 1 SECOND |
+-----+
| 1998-01-01 00:00:00                        |
+-----+
1 row in set (0.02 sec)
```

```
mysql> SELECT INTERVAL 1 DAY + '1997-12-31';
```

```
+-----+
| INTERVAL 1 DAY + '1997-12-31' |
+-----+
| 1998-01-01                    |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT '1998-01-01' - INTERVAL 1 SECOND;
```

```
+-----+
| '1998-01-01' - INTERVAL 1 SECOND |
+-----+
| 1997-12-31 23:59:59              |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT DATE_ADD('1997-12-31 23:59:59',
-> INTERVAL 1 SECOND);
```

```
+-----+
| DATE_ADD('1997-12-31 23:59:59', INTERVAL 1 SECOND) |
+-----+
| 1998-01-01 00:00:00                                |
+-----+
1 row in set (0.00 sec)
```

```

mysql> SELECT DATE_ADD('1997-12-31 23:59:59',
-> INTERVAL 1 DAY);
+-----+
| DATE_ADD('1997-12-31 23:59:59', INTERVAL 1 DAY) |
+-----+
| 1998-01-01 23:59:59                               |
+-----+
1 row in set (0.00 sec)

mysql> SELECT DATE_ADD('1997-12-31 23:59:59',
-> INTERVAL '1:1' MINUTE_SECOND);
+-----+
| DATE_ADD('1997-12-31 23:59:59', INTERVAL '1:1' MINUTE_SECOND) |
+-----+
| 1998-01-01 00:01:00                                           |
+-----+
1 row in set (0.00 sec)

mysql> SELECT DATE_SUB('1998-01-01 00:00:00',
-> INTERVAL '1 1:1:1' DAY_SECOND);
+-----+
| DATE_SUB('1998-01-01 00:00:00', INTERVAL '1 1:1:1' DAY_SECOND) |
+-----+
| 1997-12-30 22:58:59                                           |
+-----+
1 row in set (0.00 sec)

mysql> SELECT DATE_ADD('1998-01-01 00:00:00',
-> INTERVAL '-1 10' DAY_HOUR);
+-----+
| DATE_ADD('1998-01-01 00:00:00', INTERVAL '-1 10' DAY_HOUR) |
+-----+
| 1997-12-30 14:00:00                                           |
+-----+
1 row in set (0.00 sec)

mysql> SELECT DATE_SUB('1998-01-02', INTERVAL 31 DAY);
+-----+
| DATE_SUB('1998-01-02', INTERVAL 31 DAY) |
+-----+
| 1997-12-02                               |
+-----+
1 row in set (0.00 sec)

mysql> SELECT DATE_ADD('1992-12-31 23:59:59.000002',
-> INTERVAL '1.999999' SECOND_MICROSECOND);
-> '1993-01-01 00:00:01.000001'

```

Si se especifica un valor de intervalo demasiado corto (que no incluye todas las partes del intervalo que se esperan dada la palabra clave en tipo), MySQL asume que se han perdido las partes de la izquierda del valor de intervalo. Por ejemplo, si se especifica un tipo *DAY\_SECOND*, se espera que el valor de expr contenga las partes correspondientes a días, horas, minutos y segundos. Si se especifica un valor como '1:10', MySQL asume que las partes de días y horas se han perdido y el valor representa minutos y segundos. En otras palabras, '1:10' *DAY\_SECOND* se interpreta del mismo modo que si se hubiese usado '1:10' *MINUTE\_SECOND*. Esto es análogo al modo en que MySQL interpreta valores *TIME* que representan intervalos de tiempo en lugar de tiempo con respecto al día. Si se suma o resta desde un valor de fecha algo que contiene la parte de la hora, el resultado se convierte automáticamente a un valor de fecha y hora:

```
mysql> SELECT DATE_ADD('1999-01-01', INTERVAL 1 DAY);
      -> '1999-01-02'
mysql> SELECT DATE_ADD('1999-01-01', INTERVAL 1 HOUR);
      -> '1999-01-01 01:00:00'
```

Se se usan fechas mal formadas, el resultado es NULL. Si se suma *MONTH*, *YEAR\_MONTH* o *YEAR* y la fecha resultado contiene un día que es mayor que el máximo para el nuevo mes, el día se ajusta al máximo para el mes nuevo:

```
mysql> SELECT DATE_ADD('1998-01-30', interval 1 month);
      -> '1998-02-28'
```

Notar que para el ejemplo anterior la palabra clave *INTERVAL* y el especificador de tipo no son case-sensitive.

# DATE\_FORMAT()

```
DATE_FORMAT( fecha , formato )
```

Formatea el valor de fecha de acuerdo con la cadena de formato. Se pueden usar los siguientes especificadores para la cadena de formato:

Especificador	Descripción
%M	Nombre del mes (January..December)
%W	Nombre de día (Sunday..Saturday)
%D	Día del mes con sufijo en inglés (0th, 1st, 2nd, 3rd, etc.)
%Y	Año, numérico con 4 dígitos
%y	Año, numérico con 2 dígitos
%X	Año para la semana donde el domingo es el primer día de la semana, numérico de 4 dígitos; usado junto con %V
%x	Año para la semana donde el lunes es el primer día de la semana, numérico de 4 dígitos; usado junto con %v
%a	Nombre de día de semana abreviado (Sun..Sat)
%d	Día del mes, numérico (00..31)
%e	Día del mes, numérico (0..31)
%m	Mes, numérico (00..12)
%c	Mes, numérico (0..12)
%b	Nombre del mes abreviado (Jan..Dec)
%j	Día del año (001..366)
%H	Hora (00..23)
%k	Hora (0..23)
%h	Hora (01..12)
%I	Hora (01..12)
%l	Hora (1..12)
%i	Minutos, numérico (00..59)
%r	Tiempo, 12-horas (hh:mm:ss seguido por AM o PM)

%T	Tiempo, 24-horas (hh:mm:ss)
%S	Segundos (00..59)
%s	Segundos (00..59)
%f	Microsegundos (000000..999999)
%p	AM o PM
%w	Día de la semana (0=Sunday..6=Saturday)
%U	Semana (00..53), donde el domingo es el primer día de la semana
%u	Semana (00..53), donde el lunes es el primer día de la semana
%V	Semana (01..53), donde el domingo es el primer día de la semana; usado con %X
%v	Semana (01..53), donde el lunes es el primer día de la semana; usado con %X
%%	Un '%' literal.

El resto de los caracteres se copian tal cual al resultado sin interpretación. El especificador de formato %f está disponible desde MySQL 4.1.1. Desde MySQL 3.23, se requiere el carácter '%' antes de los caracteres de especificación de formato. En versiones de MySQL más antiguas, '%' era opcional. El motivo por el que los rangos de meses y días empiecen en cero es porque MySQL permite almacenar fechas incompletas como '2004-00-00', desde MySQL 3.23.

```
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00', '%W %M %Y');
+-----+
| DATE_FORMAT('1997-10-04 22:23:00', '%W %M %Y') |
+-----+
| Saturday October 1997                          |
+-----+
1 row in set (0.05 sec)
```

```
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00', '%H:%i:%s');
+-----+
| DATE_FORMAT('1997-10-04 22:23:00', '%H:%i:%s') |
+-----+
| 22:23:00                                         |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00',
->                        '%D %y %a %d %m %b %j');
+-----+
| DATE_FORMAT('1997-10-04 22:23:00',            |
|                        '%D %y %a %d %m %b %j') |
+-----+
```

## DATE\_FORMAT

```
| 4th 97 Sat 04 10 Oct 277 |
```

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

```
1 row in set (0.00 sec)
```

```
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00',  
->                        '%H %k %I %r %T %S %w');
```

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

```
| DATE_FORMAT('1997-10-04 22:23:00',  
              '%H %k %I %r %T %S %w') |
```

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

```
| 22 22 10 10:23:00 PM 22:23:00 00 6 |
```

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

```
1 row in set (0.00 sec)
```

```
mysql> SELECT DATE_FORMAT('1999-01-01', '%X %V');
```

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

```
| DATE_FORMAT('1999-01-01', '%X %V') |
```

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

```
| 1998 52 |
```

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

```
1 row in set (0.00 sec)
```

# DAYOFMONTH()

## DAY()

```
DAYOFMONTH(date)  
DAY(date)
```

Devuelve el día del mes para la fecha dada, en el rango de 1 a 31:

```
mysql> SELECT DAYOFMONTH('1998-02-03');  
+-----+  
| DAYOFMONTH('1998-02-03') |  
+-----+  
|                          3 |  
+-----+  
1 row in set (0.00 sec)
```

**DAY()** es un sinónimo de **DAYOFMONTH()**. Está disponible desde MySQL 4.1.1.

# DAYNAME()

```
DAYNAME(date)
```

Devuelve el nombre del día de la semana para una fecha:

```
mysql> SELECT DAYNAME('1998-02-05');
+-----+
| DAYNAME('1998-02-05') |
+-----+
| Thursday                |
+-----+
1 row in set (0.02 sec)
```

# DAYOFWEEK()

```
DAYOFWEEK(date)
```

Devuelve el índice del día de la semana para una fecha dada (1 = Sunday, 2 = Monday, ... 7 = Saturday). Estos valores de índice corresponden al estándar de ODBC.

```
mysql> SELECT DAYOFWEEK('1998-02-03');
+-----+
| DAYOFWEEK('1998-02-03') |
+-----+
|                          3 |
+-----+
1 row in set (0.00 sec)
```

# DAYOFYEAR()

```
DAYOFYEAR(date)
```

Devuelve el día del año para la fecha dada, en el rango de 1 a 366:

```
mysql> SELECT DAYOFYEAR('1998-02-03');
+-----+
| DAYOFYEAR('1998-02-03') |
+-----+
|                          34 |
+-----+
1 row in set (0.00 sec)
```

# DECODE

```
DECODE(crypt_str,pass_str)
```

Descripta la cadena encriptada `crypt_str` usando como contraseña `pass_str`. `crypt_str` debe ser una cadena devuelta por [ENCODE\(\)](#).

# DEFAULT

```
DEFAULT(col_name)
```

Devuelve el valor por defecto para una columna de una tabla. A partir de MySQL 5.0.2, se obtiene un error si la columna no tiene un valor por defecto.

```
mysql> UPDATE t SET i = DEFAULT(i)+1 WHERE id < 100;
```

**DEFAULT()** fue añadida en MySQL 4.1.0.

# DEGREES

DEGREES(X)

Devuelve el argumento X, convertido de radianes a grados:

```
mysql> SELECT DEGREES(PI());
+-----+
| DEGREES(PI()) |
+-----+
|           180 |
+-----+
1 row in set (0.00 sec)
```

# DES\_DECRYPT

```
DES_DECRYPT(string_to_decrypt [, key_string])
```

Desencripta una cadena encriptada con [DES\\_ENCRYPT\(\)](#). Esta función sólo funciona si MySQL ha sido configurado para soportar SSL. Si no se proporciona el argumento `key_string`, **DES\_DECRYPT()** examina el primer byte de la cadena encriptada para determinar el número de clave DES que se usó en la cadena original encriptada, a continuación lee la clave desde el fichero de claves `des` para desencriptar el mensaje. Para que esto funcione, el usuario debe poseer el privilegio `SUPER`. Si se proporciona un argumento `key_string`, esa cadena se usa como clave para desencriptar el mensaje. Si cadena `string_to_decrypt` parece no estar encriptada, MySQL devolverá la cadena `string_to_decrypt` dada. En caso de error, la función devuelve `NULL`.

# DES\_ENCRYPT

```
DES_ENCRYPT(string_to_encrypt [, (key_number | key_string) ] )
```

Encripta la cadena con la clave dada usando el algoritmo Triple-DES. Esta función sólo funciona si MySQL fue configurado con soporte SSL. La clave de encriptado a usar se elige del modo siguiente:

Argumento	Descripción
Sólo un argumento	Se usa la primera clave del fichero de claves des.
key number	Se usa la clave dada (0-9) del fichero de claves des.
key string	Se usa la cadena para encriptar.

La cadena resultante será una cadena binaria donde el primer carácter será **CHAR(128 | key\_number)**. El valor 128 se añade para hacer más fácil reconocer una clave de encriptado. Si se usa una cadena como clave, el key\_number será 127. En caso de error, la función devuelve NULL. La longitud de la cadena resultante será  $new\_length = org\_length + (8 - (org\_length \% 8)) + 1$ . El formato del fichero de claves des es el siguiente:

```
key_number des_key_string
key_number des_key_string
```

Cada key\_number debe ser un número en el rango de 0 a 9. Las líneas del fichero pueden estar en cualquier orden. des\_key\_string es la cadena que se usará para encriptar el mensaje. Entre el número y la clave debe haber al menos un espacio. La primera clave es la clave por defecto que se usará si no se especifica ningún argumento de clave en **DES\_ENCRYPT()**. Se puede indicar a MySQL que lea nuevos valores de claves desde el fichero usando el comando **FLUSH DES KEY FILE**. Para esto se necesita el privilegio Reload\_priv. Una de las ventajas de tener un conjunto de claves por defecto es que proporciona a la aplicaciones una forma de verificar la existencia de valores de columna encriptados, sin pedir del usuario final el derecho a desencriptar esos valores.

```
mysql> SELECT customer_address FROM customer_table WHERE
        crypted_credit_card = DES_ENCRYPT("credit_card_number");
```

# DIV

DIV

División entera. Es similar a [FLOOR\(\)](#) pero seguro con valores *BIGINT*.

```
mysql> SELECT 5 DIV 2  
      -> 2
```

**DIV** es nueva en MySQL 4.1.0.

# ELT()

```
ELT(N, str1, str2, str3, ...)
```

Devuelve str1 si N = 1, str2 si N = 2, y sucesivamente. Devuelve NULL si N es menor que 1 o mayor que el número de argumentos. **ELT()** es el complemento de **FIELD()**:

```
mysql> SELECT ELT(1, 'ej', 'Heja', 'hej', 'foo');
+-----+
| ELT(1, 'ej', 'Heja', 'hej', 'foo') |
+-----+
| ej                                     |
+-----+
1 row in set (0.01 sec)
```

```
mysql> SELECT ELT(4, 'ej', 'Heja', 'hej', 'foo');
+-----+
| ELT(4, 'ej', 'Heja', 'hej', 'foo') |
+-----+
| foo                                   |
+-----+
1 row in set (0.00 sec)
```

# ENCODE

```
ENCODE(str,pass_str)
```

Encripta la cadena `str` usando como contraseña `pass_str`. Para desencriptar el resultado usar [DECODE\(\)](#). El resultado es una cadena binaria de la misma longitud que `string`. Si se quiere almacenar el resultado en una columna, se debe usar una columna de tipo *BLOB*.

# ENCRYPT

```
ENCRYPT(str[,salt])
```

Encripta str usando la llamada del sistema **Unix** *crypt()*. El argumento salt debe ser una cadena con dos caracteres. (Desde MySQL 3.22.16, salt puede ser más larga de dos caracteres.)

```
mysql> SELECT ENCRYPT("hello");  
-> 'VxuFAJXVARROc'
```

**ENCRYPT()** ignora todos menos los primeros 8 caracteres de str, al menos en algunos sistemas. Este comportamiento viene determinado por la implementación de la llamada de sistema *crypt()* subyacente. Si *crypt()* no está disponible en el sistema, **ENCRYPT()** siempre devuelve NULL. Debido a esto se recomienda el uso de [MD5\(\)](#) o de [SHA1\(\)](#) en su lugar; estas dos funciones existen en todas las plataformas.

# EXP

EXP(X)

Devuelve el valor del número e (la base de los logaritmos naturales) elevado a la potencia X:

```
mysql> SELECT EXP(2);
+-----+
| EXP(2) |
+-----+
| 7.389056 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT EXP(-2);
+-----+
| EXP(-2) |
+-----+
| 0.135335 |
+-----+
1 row in set (0.00 sec)
```

# EXPORT\_SET()

```
EXPORT_SET(bits,on,off,[separator,[number_of_bits]])
```

Devuelve una cadena donde para todos los bits activos en 'bits', se obtiene una cadena 'on' y para los inactivos se obtiene una cadena 'off'. Cada cadena se separa con 'separator' (por defecto ',') y sólo 'number\_of\_bits' (por defecto 64) de 'bits' se usan:

```
mysql> SELECT EXPORT_SET(5,'Y','N',' ',4);
+-----+
| EXPORT_SET(5,'Y','N',' ',4) |
+-----+
| Y,N,Y,N                    |
+-----+
1 row in set (0.00 sec)
```

# EXTRACT()

```
EXTRACT(type FROM date)
```

La función **EXTRACT()** usa los mismos tipos de especificadores de intervalos que [DATE\\_ADD\(\)](#) o [DATE\\_SUB\(\)](#), pero extrae partes de la fecha en lugar de realizar aritmética de fechas.

```
mysql>mysql> SELECT EXTRACT(YEAR FROM "1999-07-02");
+-----+
| EXTRACT(YEAR FROM "1999-07-02") |
+-----+
|                               1999 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT EXTRACT(YEAR_MONTH FROM "1999-07-02 01:02:03");
+-----+
| EXTRACT(YEAR_MONTH FROM "1999-07-02 01:02:03") |
+-----+
|                               199907 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT EXTRACT(DAY_MINUTE FROM "1999-07-02 01:02:03");
+-----+
| EXTRACT(DAY_MINUTE FROM "1999-07-02 01:02:03") |
+-----+
|                               20102 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT EXTRACT(MICROSECOND FROM "2003-01-02 10:30:00.00123");
-> 123
```

# FIELD()

```
FIELD(str,str1,str2,str3,...)
```

Devuelve el índice de 'str' en la lista 'str1', 'str2', 'str3', .... Devuelve 0 si 'str' no se encuentra. **FIELD()** es el complemento de [ELT\(\)](#):

```
mysql> SELECT FIELD('ej', 'Hej', 'ej', 'Heja', 'hej', 'foo');
+-----+
| FIELD('ej', 'Hej', 'ej', 'Heja', 'hej', 'foo') |
+-----+
|                                               2 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT FIELD('fo', 'Hej', 'ej', 'Heja', 'hej', 'foo');
+-----+
| FIELD('fo', 'Hej', 'ej', 'Heja', 'hej', 'foo') |
+-----+
|                                               0 |
+-----+
1 row in set (0.00 sec)
```

# FIND\_IN\_SET()

```
FIND_IN_SET(str, strlist)
```

Devuelve un valor de 1 a N si la cadena *str* está en la lista *strlist* que consiste en N subcadenas. Una lista de cadenas es una cadena compuesta por subcadenas separadas por caracteres ','. Si el primer argumento es una cadena constante y la segunda es una columna de tipo *SET*, la función **FIND\_IN\_SET()** está optimizada para usar aritmética de bits. Devuelve 0 si *str* no está en *strlist* o si *strlist* es una cadena vacía. Devuelve NULL si cualquiera de los argumentos es NULL. Esta función no funcionará adecuadamente si el primer argumento contiene una coma ',':

```
mysql> SELECT FIND_IN_SET('b', 'a,b,c,d');
+-----+
| FIND_IN_SET('b', 'a,b,c,d') |
+-----+
|                               2 |
+-----+
1 row in set (0.02 sec)
```

# FLOOR

FLOOR(X)

Devuelve el entero más grande inferior o igual a X:

```
mysql> SELECT FLOOR(1.23);
```

```
+-----+  
| FLOOR(1.23) |  
+-----+  
|           1 |  
+-----+
```

```
1 row in set (0.00 sec)
```

```
mysql> SELECT FLOOR(-1.23);
```

```
+-----+  
| FLOOR(-1.23) |  
+-----+  
|           -2 |  
+-----+
```

```
1 row in set (0.00 sec)
```

El valor de retorno se convierte a *BIGINT*.

# FORMAT

```
FORMAT(X,D)
```

Formatea el número según la plantilla '#,###,###.##', redondeando a D decimales, y devuelve el resultado como una cadena. Si D es 0, el resultado no tendrá punto decimal ni parte fraccionaria:

```
mysql> SELECT FORMAT(12332.123456, 4);
```

```
+-----+
| FORMAT(12332.123456, 4) |
+-----+
| 12,332.1235           |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT FORMAT(12332.1,4);
```

```
+-----+
| FORMAT(12332.1,4) |
+-----+
| 12,332.1000       |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT FORMAT(12332.2,0);
```

```
+-----+
| FORMAT(12332.2,0) |
+-----+
| 12,332            |
+-----+
1 row in set (0.00 sec)
```

# FOUND\_ROWS

```
FOUND_ROWS()
```

Una sentencia [SELECT](#) puede incluir una cláusula *LIMIT* para restringir el número de filas que el servidor devuelve al cliente. En algunos casos, es posible que se quiera conocer cuántas filas se hubiesen obtenido sin la cláusula *LIMIT*, pero sin ejecutar la sentencia de nuevo. Para obtener este número, hay que incluir la opción *SQL\_CALC\_FOUND\_ROWS* en la sentencia [SELECT](#), y después invocar la función **FOUND\_ROWS()**:

```
mysql> SELECT SQL_CALC_FOUND_ROWS * FROM tbl_name
        WHERE id > 100 LIMIT 10;
mysql> SELECT FOUND_ROWS();
```

El segundo [SELECT](#) devolverá un número que indica cuántas filas hubiera devuelto el primer [SELECT](#) si no se hubiese usado la cláusula *LIMIT*. (Si la sentencia [SELECT](#) no incluye la opción *SQL\_CALC\_FOUND\_ROWS*, entonces **FOUND\_ROWS()** podrá devolver un valor diferente cuando se usa *LIMIT* y cuando no se usa.) Si se usa [SELECT SQL\\_CALC\\_FOUND\\_ROWS ...](#) MySQL debe calcular cuántas filas hay en el conjunto de resultados completo. Sin embargo, esto es más rápido que ejecutar una nueva consulta sin el *LIMIT*, ya que no es necesario enviar el conjunto de resultados al cliente. *SQL\_CALC\_FOUND\_ROWS* y **FOUND\_ROWS()** suelen usarse en situaciones en las que se quiere restringir el número de filas que devuelva una consulta, pero también se quiere determinar el número de filas en el conjunto de resultados completo sin ejecutar la consulta de nuevo. Un ejemplo es un script web que presente una muestra paginada conteniendo enlaces a páginas que muestran otras secciones del resultado de una búsqueda. Usando **FOUND\_ROWS()** es posible determinar cuántas de esas páginas son necesarias para mostrar el resto de los resultados. El uso de *SQL\_CALC\_FOUND\_ROWS* y **FOUND\_ROWS()** es más complejo en el caso de consultas [UNION](#) que para sentencias sencillas [SELECT](#), porque *LIMIT* puede ocurrir en muchos lugares en una [UNION](#). Debe ser aplicado a sentencias [SELECT](#) individuales en la [UNION](#), o globalmente al resultado de la *UNION* completo. El objetivo de *SQL\_CALC\_FOUND\_ROWS* para [UNION](#) es que devuelva el número de filas que serían devueltas sin un *LIMIT* global. Las condiciones para usar *SQL\_CALC\_FOUND\_ROWS* con [UNION](#) son:

- La palabra clave *SQL\_CALC\_FOUND\_ROWS* debe aparecer en el primer [SELECT](#) de la [UNION](#).
- El valor de **FOUND\_ROWS()** es exacto sólo si se usa [UNION ALL](#). Si se usa [UNION](#) sin *ALL*, se eliminan los duplicados y el valor de **FOUND\_ROWS()** sólo es aproximado.
- Si no hay una cláusula *LIMIT* en la [UNION](#), *SQL\_CALC\_FOUND\_ROWS* se ignora y devuelve

el número de filas en la tabla temporal que se crearon al procesar UNION.

*SQL\_CALC\_FOUND\_ROWS* y **FOUND\_ROWS()** están disponibles desde la versión 4.0.0 de MySQL.

# FROM\_DAYS()

```
FROM_DAYS(N)
```

Dado un número de día *N*, devuelve un valor de fecha *DATE*:

```
mysql> SELECT FROM_DAYS(729669);  
      -> '1997-10-07'
```

**FROM\_DAYS()** no está diseñada para usarse con valores anteriores al comienzo del calendario Gregoriano (1582), porque no tiene en cuenta los días que se perdieron cuando el calendario se cambió.

# FROM\_UNIXTIME()

```
FROM_UNIXTIME(unix_timestamp)
FROM_UNIXTIME(unix_timestamp,formato)
```

Devuelve una representación del argumento *unix\_timestamp* como un valor en el formato 'YYYY-MM-DD HH:MM:SS' o YYYYMMDDHHMMSS, dependiendo de si la función se usa en un contexto de cadena o numérico:

```
mysql> SELECT FROM_UNIXTIME(875996580);
+-----+
| FROM_UNIXTIME(875996580) |
+-----+
| 1997-10-04 22:23:00      |
+-----+
1 row in set (0.02 sec)

mysql> SELECT FROM_UNIXTIME(875996580) + 0;
+-----+
| FROM_UNIXTIME(875996580) + 0 |
+-----+
|                19971004222300 |
+-----+
1 row in set (0.02 sec)
```

Si se proporciona un formato, el resultado se formatea de acuerdo con la cadena de formato. *format* puede contener los mismos especificadores que se listan en la entrada de la función [DATE\\_FORMAT\(\)](#):

```
mysql> SELECT FROM_UNIXTIME(UNIX_TIMESTAMP(),
->                          '%Y %D %M %h:%i:%s %x');
+-----+
| FROM_UNIXTIME(UNIX_TIMESTAMP(), '%Y %D %M %h:%i:%s %x') |
+-----+
| 2003 26th December 08:12:24 2003                          |
+-----+
1 row in set (0.02 sec)
```

# GET\_FORMAT()

```
GET_FORMAT( DATE | TIME | TIMESTAMP, 'EUR' | 'USA' | 'JIS' | 'ISO' |
'INTERNAL' )
```

Devuelve una cadena de formato. Esta función es frecuente en combinación con las funciones [DATE\\_FORMAT\(\)](#) y [STR\\_TO\\_DATE\(\)](#). Los tres posibles valores para el primer argumento y los cinco para el segundo implican quince posibles cadenas de formato (para los especificadores usados, ver la tabla en la descripción de la función [DATE\\_FORMAT\(\)](#)):

Llamada a función	Resultado
GET_FORMAT(DATE,'USA')	'%m.%d.%Y'
GET_FORMAT(DATE,'JIS')	'%Y-%m-%d'
GET_FORMAT(DATE,'ISO')	'%Y-%m-%d'
GET_FORMAT(DATE,'EUR')	'%d.%m.%Y'
GET_FORMAT(DATE,'INTERNAL')	'%Y%m%d'
GET_FORMAT(TIMESTAMP,'USA')	'%Y-%m-%d-%H.%i.%s'
GET_FORMAT(TIMESTAMP,'JIS')	'%Y-%m-%d %H:%i:%s'
GET_FORMAT(TIMESTAMP,'ISO')	'%Y-%m-%d %H:%i:%s'
GET_FORMAT(TIMESTAMP,'EUR')	'%Y-%m-%d-%H.%i.%s'
GET_FORMAT(TIMESTAMP,'INTERNAL')	'%Y%m%d%H%i%s'
GET_FORMAT(TIME,'USA')	'%h:%i:%s %p'
GET_FORMAT(TIME,'JIS')	'%H:%i:%s'
GET_FORMAT(TIME,'ISO')	'%H:%i:%s'
GET_FORMAT(TIME,'EUR')	'%H.%i.%S'
GET_FORMAT(TIME,'INTERNAL')	'%H%i%s'

El formato ISO se refiere a ISO 9075, no a ISO 8601.

```
mysql> SELECT DATE_FORMAT('2003-10-03', GET_FORMAT(DATE, 'EUR'));
-> '03.10.2003'
```

```
mysql> SELECT STR_TO_DATE('10.31.2003', GET_FORMAT(DATE, 'USA'));  
-> 2003-10-31
```

**GET\_FORMAT()** está disponible a partir de MySQL 4.1.1.

# GET\_LOCK

```
GET_LOCK(str,timeout)
```

Intenta obtener un bloqueo con el nombre dado por la cadena `str`, con un tiempo límite de `timeout` segundos. Devuelve 1 si se ha obtenido el bloqueo, 0 si no se ha obtenido en el tiempo indicado (por ejemplo, porque otro cliente ha bloqueado ya el nombre), o NULL si ha habido un error (como falta de memoria o si el hilo fue matado por *mysqladmin kill*). Un bloqueo se libera cuando se ejecuta la función [RELEASE\\_LOCK\(\)](#), se ejecuta un nuevo **GET\_LOCK()** o el hilo termina (ya sea normal o anormalmente). Esta función se puede usar para implementar bloqueos de aplicación o para simular bloqueos de registro. Los nombres se bloquean en bases del servidor. Si un nombre ha sido bloqueado por un cliente, **GET\_LOCK()** bloquea cualquier petición de otro cliente para bloquear el mismo nombre. Esto permite que clientes que se ponen de acuerdo para bloquear un nombre dado usar el mismo nombre para realizar un bloqueo coordinado:

```
mysql> SELECT GET_LOCK("lock1",10);
      -> 1
mysql> SELECT IS_FREE_LOCK("lock2");
      -> 1
mysql> SELECT GET_LOCK("lock2",10);
      -> 1
mysql> SELECT RELEASE_LOCK("lock2");
      -> 1
mysql> SELECT RELEASE_LOCK("lock1");
      -> NULL
```

El segunda llamada a [RELEASE\\_LOCK\(\)](#) devuelve NULL porque el bloqueo "lock1" se ha liberado de forma automática por la segunda llamada a **GET\_LOCK()**.

# GREATEST

```
GREATEST(X,Y,...)
```

Devuelve el argumento mayor. Los argumentos son comparados usando las mismas reglas que para [LEAST\(\)](#).

```
mysql> SELECT GREATEST(2,0);
+-----+
| GREATEST(2,0) |
+-----+
|                2 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT GREATEST(34.0,3.0,5.0,767.0);
+-----+
| GREATEST(34.0,3.0,5.0,767.0) |
+-----+
|                               767.0 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT GREATEST("B","A","C");
+-----+
| GREATEST("B","A","C") |
+-----+
| C                       |
+-----+
1 row in set (0.00 sec)
```

En versiones de MySQL anteriores a 3.22.5, se puede usar [MAX\(\)](#) en lugar de **GREATEST**.

# GROUP\_CONCAT

```
GROUP_CONCAT(expr)
```

Sintaxis completa:

```
GROUP_CONCAT([DISTINCT] expr [,expr ...]
             [ORDER BY {unsigned_integer | col_name | formula} [ASC | DESC] [,
             col ...]])
             [SEPARATOR str_val])
```

Esta función se añadió en MySQL 4.1. Devuelve una cadena con la concatenación de los valores del grupo:

```
mysql> SELECT student_name,
->          GROUP_CONCAT(test_score)
->          FROM student
->          GROUP BY student_name;
```

O:

```
mysql> SELECT student_name,
->          GROUP_CONCAT(DISTINCT test_score
->                        ORDER BY test_score DESC SEPARATOR " ")
->          FROM student
->          GROUP BY student_name;
```

En MySQL se pueden obtener los valores concatenados de combinaciones de expresiones. Se pueden eliminar valores duplicados usando *DISTINCT*. Si se desea ordenar los valores del resultado se puede usar la cláusula *ORDER BY*. Para ordenar en orden inverso, añadir la palabra clave *DESC* (descendente) al nombre de la columna por la que se está ordenando en la cláusula *ORDER BY*. Por defecto, el orden es ascendente; que se puede especificar explícitamente usando la palabra clave *ASC*. *SEPARATOR* es el valor de cadena que se insertará entre los valores del resultado. Por defecto es una coma (","). Se puede eliminar el separador por completo especificando *SEPARATOR ""*. También se puede limitar la longitud máxima con la variable *group\_concat\_max\_len* en la configuración. La sintaxis para hacerlo durante la ejecución de MySQL:

```
SET [SESSION | GLOBAL] group_concat_max_len = unsigned_integer;
```

Si se ha asignado una longitud máxima, el resultado se truncará a esa longitud. La función **GROUP\_CONCAT()** es una implementación mejorada de la función básica **LIST()** soportada por **Sybase SQL Anywhere**. **GROUP\_CONCAT()** mantiene compatibilidad con la extremadamente limitada funcionalidad de **LIST()**, si sólo se especifica una columna y ninguna otra opción. **LIST()** tiene un modo de orden por defecto.

Si se usa una función de grupo en una sentencia que contenga la cláusula *GROUP BY*, equivale a agrupar todas las filas.

# HEX()

```
HEX(N_or_S)
```

Si N\_OR\_S es un número, devuelve una cadena que representa el valor hexadecimal de N, donde N es un número longlong (BIGINT). Es equivalente a [CONV\(N,10,16\)](#). Si N\_OR\_S es una cadena, devuelve una cadena hexadecimal de N\_OR\_S donde cada carácter en N\_OR\_S se convierte a dos dígitos hexadecimales. Esto es la inversa de las cadenas 0xff.

```
mysql> SELECT HEX(255);
```

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

```
| HEX(255) |
```

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

```
| FF      |
```

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

```
1 row in set (0.03 sec)
```

```
mysql> SELECT HEX("abc");
```

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

```
| HEX("abc") |
```

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

```
| 616263    |
```

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

```
1 row in set (0.00 sec)
```

```
mysql> SELECT 0x616263;
```

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

```
| 0x616263 |
```

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

```
| abc      |
```

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

```
1 row in set (0.00 sec)
```

```
mysql>
```

# hour()

hour(time)

Devuelve la hora para time. El rango del valor retornado puede ser de 0 a 23 para valores de horas correspondientes al día:

```
mysql> SELECT HOUR('10:05:03');
+-----+
| HOUR('10:05:03') |
+-----+
|                10 |
+-----+
1 row in set (0.00 sec)
```

Sin embargo, el rango de valores de *TIME* es mucho mayor, de modo que **hour** puede devolver valores mayores de 23:

```
mysql> SELECT HOUR('272:59:59');
+-----+
| HOUR('272:59:59') |
+-----+
|                272 |
+-----+
1 row in set (0.00 sec)
```

# IF

```
IF(expr1,expr2,expr3)
```

Si la `expr1` es `TRUE` (`expr1 <> 0` and `expr1 <> NULL`) entonces **IF()** devuelve `expr2`, en caso contrario, devolverá `expr3`. **IF()** devuelve un valor numérico o una cadena, dependiendo del contexto en el que se use.

```
mysql> SELECT IF(1>2,2,3);
      -> 3
mysql> SELECT IF(1<2,'yes','no');
      -> 'yes'
mysql> SELECT IF(STRCMP('test','test1'),'no','yes');
      -> 'no'
```

Si sólo `expr2` o `expr3` es explícitamente `NULL`, el tipo del resultado de la función **IF()** es el de la expresión no nula. (Este comportamiento es nuevo en MySQL 4.0.3.) `expr1` se evalúa como un valor entero, lo que significa que si se están comprobando valores en coma flotante o cadenas, se debe usar siempre una operación de comparación.

```
mysql> SELECT IF(0.1,1,0);
      -> 0
mysql> SELECT IF(0.1<>0,1,0);
      -> 1
```

En el primer caso mostrado, **IF(0.1)** devuelve 0 porque 0.1 se convierte a un valor entero, resultando la verificación **IF(0)**. Esto no es lo que probablemente se esperaba. En el segundo caso, la comparación comprueba el valor de punto flotante original para comprobar si es distinto de cero. El resultado de la comparación se usa como un entero. El tipo del valor de retorno por defecto para **IF()** (lo cual puede ser importante cuando sea almacenado en una tabla temporal) se calcula en MySQL 3.23 como sigue:

Expresión	Valor de retorno
<code>expr2</code> o <code>expr3</code> devuelve una cadena	cadena
<code>expr2</code> o <code>expr3</code> devuelve un valor en coma flotante	coma flotante
<code>expr2</code> o <code>expr3</code> devuelve un entero	entero

IF

Si `expr2` y `expr3` son cadenas, el resultado será sensible al tipo si cualquiera de las cadenas lo es (a partir de MySQL 3.23.51).

# IFNULL

```
IFNULL(expr1,expr2)
```

Si *expr1* no es *NULL*, **IFNULL()** devuelve *expr1*, en caso contrario, devuelve *expr2*. **IFNULL()** devuelve un valor numérico o una cadena, dependiendo del contexto en el que se use.

```
mysql> SELECT IFNULL(1,0);
      -> 1
mysql> SELECT IFNULL(NULL,10);
      -> 10
mysql> SELECT IFNULL(1/0,10);
      -> 10
mysql> SELECT IFNULL(1/0,'yes');
      -> 'yes'
```

En MySQL 4.0.6 y posteriores, el valor de resultado por defecto de **IFNULL(expr1,expr2)** es la más "general" de las dos expresiones, en el orden *STRING*, *REAL* o *INTEGER*. La diferencia con versiones de MySQL anteriores es generalmente más notable cuando se crea una tabla basada en expresiones o MySQL no almacena internamente el valor de un **IFNULL()** en una tabla temporal.

```
CREATE TABLE tmp SELECT IFNULL(1,'test') AS test;
```

Desde MySQL 4.0.6, el tipo para la columna 'test' es *CHAR(4)*, mientras que en versiones anteriores el tipo sería *BIGINT*.

# INET\_ATON

```
INET_ATON(expr)
```

Dada la dirección de red en formato de cuarteto con puntos como una cadena, devuelve un entero que representa el valor numérico de la dirección. Las direcciones pueden ser de 4 u 8 bytes:

```
mysql> SELECT INET_ATON("209.207.224.40");
+-----+
| INET_ATON("209.207.224.40") |
+-----+
|                3520061480 |
+-----+
1 row in set (0.00 sec)
```

El número generado es siempre en el orden de bytes de red; por ejemplo el número anterior se calcula como  $209*256^3 + 207*256^2 + 224*256 + 40$ .

# INET\_NTOA

```
INET_NTOA(expr)
```

Dada una dirección de red numérica(4 o 8 bytes), devuelve la cadena que representa la dirección en formato de cuarteto separado con puntos:

```
mysql> SELECT INET_NTOA(3520061480);
+-----+
| INET_NTOA(3520061480) |
+-----+
| 209.207.224.40        |
+-----+
1 row in set (0.00 sec)
```

# INSERT()

```
INSERT(str, pos, len, newstr)
```

Devuelve la cadena str, con la subcadena que empieza en la posición pos y de len caracteres de longitud reemplazada con la cadena newstr:

```
mysql> SELECT INSERT('Quadratic', 3, 4, 'What');
+-----+
| INSERT('Quadratic', 3, 4, 'What') |
+-----+
| QuWhattic                          |
+-----+
1 row in set (0.03 sec)
```

Esta función es segura con caracteres multibyte.

# INSTR()

```
INSTR(str,substr)
```

Devuelve la posición de la primera aparición de la subcadena substr dentro de la cadena str. Es lo mismo que la forma de [LOCATE\(\)](#) con dos argumentos, excepto que los argumentos están intercambiados:

```
mysql> SELECT INSTR('foobarbar', 'bar');
```

```
+-----+
| INSTR('foobarbar', 'bar') |
+-----+
|                          4 |
+-----+
```

```
1 row in set (0.01 sec)
```

```
mysql> SELECT INSTR('xbar', 'foobar');
```

```
+-----+
| INSTR('xbar', 'foobar') |
+-----+
|                          0 |
+-----+
```

```
1 row in set (0.00 sec)
```

Esta función es segura con caracteres multibyte. En MySQL 3.23 esta función distingue mayúsculas y minúsculas, mientras que en la versión 4.0 no lo hace if cualquiera de los argumentos es una cadena binaria.

# IS\_FREE\_LOCK

```
IS_FREE_LOCK(str)
```

Verifica si el nombre de bloqueo `str` está libre para usarse (es decir, no está bloqueado). Devuelve 1 si el bloqueo está libre (nadie está usando el bloqueo), 0 si el bloqueo está en uso y NULL si hay errores (como argumentos incorrectos).

# ***IS\_USED\_LOCK***

```
IS_USED(str)
```

Verifica si el nombre de bloqueo `str` está en uso (es decir, bloqueado). Si lo está, devuelve el identificador de conexión del cliente que mantiene el bloqueo. En otro caso devuelve `NULL`.

**IS\_USED\_LOCK()** fue añadido en MySQL 4.1.0.

# LAST\_DAY()

```
LAST_DAY(date)
```

Toma un valor fecha o fecha y hora y devuelve el valor correspondiente para el último día del mes. Devuelve NULL si el argumento no es válido.

```
mysql> SELECT LAST_DAY('2003-02-05'), LAST_DAY('2004-02-05');
      -> '2003-02-28', '2004-02-29'
mysql> SELECT LAST_DAY('2004-01-01 01:01:01');
      -> '2004-01-31'
mysql> SELECT LAST_DAY('2003-03-32');
      -> NULL
```

**LAST\_DAY()** está disponible desde MySQL 4.1.1.

# LAST\_INSERT\_ID

```
LAST_INSERT_ID([expr])
```

Devuelve el último valor generado automáticamente que fue insertado en una columna *AUTO\_INCREMENT*.

```
mysql> SELECT LAST_INSERT_ID();  
-> 195
```

El último ID que fue generado se mantiene en el servidor en una base por conexión. Esto significa que el valor que devuelve la función para un cliente dado es valor *AUTO\_INCREMENT* más reciente generado por ese cliente. El valor no puede verse afectado por otros clientes, aunque generen valores *AUTO\_INCREMENT* por si mismos. Este comportamiento asegura que se puede recuperar un ID sin preocuparse por la actividad de otros clientes, y sin necesidad de bloqueos o transacciones. El valor de **LAST\_INSERT\_ID()** no cambia si se actualiza una columna *AUTO\_INCREMENT* de una fila con un valor no mágico (es decir, un valor que no es NULL ni 0). Si se insertan muchas filas al mismo tiempo con una sentencia **INSERT**, **LAST\_INSERT\_ID()** devuelve el valor para la primera fila insertada. El motivo para esto es hacer posible reproducir más fácilmente la misma sentencia **INSERT** de nuevo en algún otro servidor. Si se da un argumento *expr* a **LAST\_INSERT\_ID()**, el valor del argumento será devuelto por la función, y se asigna como siguiente valor a retornar por **LAST\_INSERT\_ID()**. Esto se puede usar para simular secuencias:

Primero crear la tabla:

```
mysql> CREATE TABLE sequence (id INT NOT NULL);  
mysql> INSERT INTO sequence VALUES (0);
```

Después la tabla se puede usar para generar secuencias de números como esta:

```
mysql> UPDATE sequence SET id=LAST_INSERT_ID(id+1);
```

Se pueden generar secuencias sin llamar a **LAST\_INSERT\_ID()**, pero al utilidad de usar la función de este modo es que el valor ID se mantiene en el servidor como el último valor generado automáticamente (seguro en multiusuario). Se puede recuperar el nuevo ID como si se recuperase cualquier valor

*AUTO\_INCREMENT* normal en MySQL. Por ejemplo, **LAST\_INSERT\_ID()** (sin argumentos) devolverá el nuevo ID. La función del API C [mysql\\_insert\\_id\(\)](#) también se puede usar para obtener el valor. [mysql\\_insert\\_id\(\)](#) sólo se actualiza después de sentencias **INSERT** y **UPDATE**, de modo que no se puede usar la función del API C para recuperar el valor para **LAST\_INSERT\_ID(expr)** después de ejecutar otra sentencia SQL como **SELECT** o **SET**.

# LOWER()

## LCASE()

```
LOWER(str)  
LCASE(str)
```

Devuelve la cadena *str* con todos los caracteres cambiados a minúsculas de acuerdo con el mapa de caracteres actual (por defecto es *ISO-8859-1 Latin1*):

```
mysql> SELECT LOWER('QUADRATICALLY');  
      -> 'quadratically'
```

**LCASE()** es sinónimo de **LOWER()**.

Esta función es segura para caracteres multibyte.

# LEAST

```
LEAST(X,Y,...)
```

Con dos o más argumentos, devuelve el menor de ellos. Los argumentos son comparados usando las reglas siguientes:

- Si el valor de retorno se usa en un contexto *INTEGER*, o si todos los argumentos son valores enteros, se comparan como enteros.
- Si el valor de retorno se usa en un contexto *REAL*, o si todos los argumentos son valores reales, se comparan como reales.
- Si cualquier argumento es una cadena case-sensitive, los argumentos se comparan como cadenas case-sensitive.
- En otros casos, los argumentos se comparan como cadenas sin tener en cuenta mayúsculas y minúsculas (case-insensitive):

```
mysql> SELECT LEAST(2,0);
+-----+
| LEAST(2,0) |
+-----+
|          0 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT LEAST(34.0,3.0,5.0,767.0);
+-----+
| LEAST(34.0,3.0,5.0,767.0) |
+-----+
|                   3.0 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT LEAST("B","A","C");
+-----+
| LEAST("B","A","C") |
+-----+
| A                   |
+-----+
1 row in set (0.00 sec)
```

En versiones de MySQL previas a 3.22.5, se puede usar [MIN\(\)](#) en lugar de **LEAST**. Hay que tener en cuenta que las reglas anteriores pueden producir resultados extraños en algunos casos límite:

```
mysql> SELECT CAST(least(3600, 9223372036854775808.0) as SIGNED);
+-----+
| CAST(least(3600, 9223372036854775808.0) as SIGNED) |
+-----+
|                                     -9223372036854775808 |
+-----+
1 row in set (0.00 sec)
```

Esto ocurre porque MySQL lee 9223372036854775808.0 en un contexto entero y la representación entera no es suficiente para almacenar el valor así que se convierte a un entero con signo.

# LEFT()

```
LEFT(cadena, longitud)
```

Devuelve los 'longitud' caracteres de la izquierda de la 'cadena':

```
mysql>mysql> SELECT LEFT('MySQL con Clase', 5);
+-----+
| LEFT('MySQL con Clase', 5) |
+-----+
| MySQL                       |
+-----+
1 row in set (0.00 sec)
```

# LENGTH()

## OCTET\_LENGTH()

```
LENGTH(str)
OCTET_LENGTH(str)
```

Devuelve la longitud de la cadena str, medida en bytes. Un carácter multibyte cuenta cómo bytes múltiples. Esto significa que para cadenas que contengan cinco caracteres de dos bytes, **LENGTH()** devuelve 10, mientras que [CHAR\\_LENGTH\(\)](#) devuelve 5.

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

**OCTET\_LENGTH()** es sinónimo de **LENGTH()**.

# LN

LN(X)

Devuelve el logaritmo natural de X:

```
mysql> SELECT LN(2);
+-----+
| LN(2) |
+-----+
| 0.693147 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT LN(-2);
+-----+
| LN(-2) |
+-----+
| NULL |
+-----+
1 row in set (0.00 sec)
```

Esta función se añadió en MySQL 4.0.3. Es sinónimo de [LOG\(X\)](#) en MySQL.

# LOAD\_FILE()

```
LOAD_FILE(file_name)
```

Lee el fichero y devuelve su contenido como una cadena. El fichero debe estar en el servidor, se debe especificar el camino completo al fichero, y se debe poseer el privilegio *FILE*. El fichero debe ser legible para todos y más pequeño que *max\_allowed\_packet*. Si el fichero no existe o no puede ser leído por alguna de las razones anteriores, la función devuelve NULL:

```
mysql> UPDATE tbl_name
        SET blob_column=LOAD_FILE("/tmp/picture")
        WHERE id=1;
```

Si no se está usando MySQL 3.23, se tendrá que hacer la lectura del fichero dentro de la aplicación y crear una sentencia **INSERT** para actualizar la base de datos con la información del fichero. Un modo de hacer esto, si se usa la librería MySQL++, se puede encontrar en <http://www.mysql.com/documentation/mysql++/mysql++-examples.html>.

# LOCATE()

## POSITION()

```
LOCATE(substr, str)
LOCATE(substr, str, pos)
POSITION(substr IN str)
```

La primera forma devuelve la posición de la primer aparición de la cadena substr dentro de la cadena str. La segunda devuelve la posición de la primera aparición de la cadena substr dentro de la cadena str, comenzando en la posición pos. Devuelve 0 si substr no está en str.

```
mysql> SELECT LOCATE('bar', 'foobarbar');
+-----+
| LOCATE('bar', 'foobarbar') |
+-----+
|                               4 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT LOCATE('xbar', 'foobar');
+-----+
| LOCATE('xbar', 'foobar') |
+-----+
|                               0 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT LOCATE('bar', 'foobarbar',5);
+-----+
| LOCATE('bar', 'foobarbar',5) |
+-----+
|                               7 |
+-----+
1 row in set (0.00 sec)
```

Esta función es segura con cadenas multibyte. En MySQL 3.23 esta función distingue entre mayúsculas y minúsculas, en la versión 4.0 sólo si cualquiera de los argumentos es una cadena binaria.

**POSITION(substr IN str)** es sinónimo de **LOCATE(substr, str)**.

# LOG

```
LOG(X)  
LOG(B,X)
```

Si se llama con un parámetro, esta función devuelve el logaritmo natural de X:

```
mysql> SELECT LOG(2);  
+-----+  
| LOG(2) |  
+-----+  
| 0.693147 |  
+-----+  
1 row in set (0.00 sec)  
  
mysql> SELECT LOG(-2);  
+-----+  
| LOG(-2) |  
+-----+  
| NULL |  
+-----+  
1 row in set (0.00 sec)
```

Si se llama con dos parámetros, devuelve el logaritmo de X para una base arbitraria B:

```
mysql> SELECT LOG(2,65536);  
+-----+  
| LOG(2,65536) |  
+-----+  
| 16.000000 |  
+-----+  
1 row in set (0.00 sec)  
  
mysql> SELECT LOG(1,100);  
+-----+  
| LOG(1,100) |  
+-----+  
| NULL |  
+-----+  
1 row in set (0.00 sec)
```

La opción de base arbitraria se añadió en MySQL 4.0.3. **LOG(B,X)** equivale a **LOG(X)/LOG(B)**.

# LOG10

LOG10(X)

Devuelve el logaritmo en base 10 de X:

```
mysql> SELECT LOG10(2);
+-----+
| LOG10(2) |
+-----+
| 0.301030 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT LOG10(100);
+-----+
| LOG10(100) |
+-----+
| 2.000000 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT LOG10(-100);
+-----+
| LOG10(-100) |
+-----+
| NULL |
+-----+
1 row in set (0.00 sec)
```

# LOG2

LOG2(X)

Devuelve el logaritmo en base 2 de X:

```
mysql> SELECT LOG2(65536);
+-----+
| LOG2(65536) |
+-----+
|    16.000000 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT LOG2(-100);
+-----+
| LOG2(-100) |
+-----+
|          NULL |
+-----+
1 row in set (0.00 sec)
```

**LOG2()** es corriente para calcular cuantos bits son necesarios para almacenar un número. Esta función se añadió en la versión 4.0.3 de MySQL. En versiones anteriores, se puede usar en su lugar **LOG(X)/LOG(2)**.

# LPAD()

```
LPAD(str, len, padstr)
```

Devuelve la cadena str, rellena a la izquierda con la cadena padstr hasta la longitud de len caracteres. Si str es más larga que len, el valor retornado se acorta hasta len caracteres.

```
mysql> SELECT LPAD('hi',4,'??');
+-----+
| LPAD('hi',4,'??') |
+-----+
| ??hi              |
+-----+
1 row in set (0.03 sec)
```

# LTRIM()

```
LTRIM(str)
```

Devuelve la cadena str con los caracteres de espacios iniciales eliminados:

```
mysql> SELECT LTRIM('  barbar');
+-----+
| LTRIM('  barbar') |
+-----+
| barbar            |
+-----+
1 row in set (0.00 sec)
```

# MAKEDATE()

```
MAKEDATE(year, dayofyear)
```

Devuelve una fecha, dados los valores del año y de día del año. dayofyear debe ser mayor que 0 o el resultado será NULL.

```
mysql> SELECT MAKEDATE(2001,31), MAKEDATE(2001,32);  
      -> '2001-01-31', '2001-02-01'  
mysql> SELECT MAKEDATE(2001,365), MAKEDATE(2004,365);  
      -> '2001-12-31', '2004-12-30'  
mysql> SELECT MAKEDATE(2001,0);  
      -> NULL
```

**MAKEDATE()** está disponible desde MySQL 4.1.1.

# MAKETIME()

```
MAKETIME(hour,minute,second)
```

Devuelve un valor de tiempo calculado a partir de los argumentos hour, minute y second.

```
mysql> SELECT MAKETIME(12,15,30);  
      -> '12:15:30'
```

**MAKETIME()** está disponible desde MySQL 4.1.1.

# MAKE\_SET()

```
MAKE_SET(bits, str1, str2, ...)
```

Devuelve un conjunto (una cadena que contiene subcadenas separadas por caracteres ',') consistente en las cadenas correspondientes a los bits activos en bits. str1 corresponde al bit 0, str2 al bit 1, etc. Las cadenas NULL en str1, str2, ... no se añaden al resultado:

```
+-----+
| MAKE_SET(1, 'a', 'b', 'c') |
+-----+
| a                          |
+-----+
1 row in set (0.00 sec)

mysql> SELECT MAKE_SET(1 | 4, 'hello', 'nice', 'world');
+-----+
| MAKE_SET(1 | 4, 'hello', 'nice', 'world') |
+-----+
| hello,world                               |
+-----+
1 row in set (0.02 sec)

mysql> SELECT MAKE_SET(0, 'a', 'b', 'c');
+-----+
| MAKE_SET(0, 'a', 'b', 'c') |
+-----+
|                               |
+-----+
1 row in set (0.00 sec)
```

# MASTER\_POS\_WAIT

```
MASTER_POS_WAIT(log_name, log_pos [, timeout])
```

Detiene hasta que el esclavo alcanza (es decir, ha leído y aplicado todas las actualizaciones hasta) la posición especificada en el diario maestro. Si la información maestra no está inicializada, o si los argumentos son incorrectos, devuelve NULL. Si el esclavo no está en ejecución, se detiene y espera hasta que sea iniciado y llegue o sobrepase la posición especificada. Si el esclavo ya ha pasado la posición especificada, regresa inmediatamente. Si se especifica un tiempo límite, timeout (nuevo en 4.0.10), se dejará de esperar cuando hayan transcurrido timeout segundos. timeout debe ser más mayor que 0; valores de timeout cero o negativos significan que no hay tiempo límite. El valor de retorno es el número de eventos del diario que se ha esperado realizar hasta la posición especificada, o NULL en caso de error, o -1 si se ha excedido el tiempo límite. Este comando es corriente para controlar la sincronización maestro/esclavo.

# MIN

# MAX

```
MIN(expr)  
MAX(expr)
```

Devuelve el valor mínimo o máximo de `expr`. **MIN()** y **MAX()** pueden tomar como argumento una cadena, en ese caso devolverán el valor de la cadena mínima o máxima.

```
mysql> SELECT student_name, MIN(test_score), MAX(test_score)  
->      FROM student  
->      GROUP BY student_name;
```

Con **MIN()**, **MAX()** y otras funciones, MySQL normalmente compara las columnas *ENUM* y *SET* por sus valores de cadena, en lugar de por sus posiciones relativas dentro del conjunto. Esto será modificado.

Si se usa una función de grupo en una sentencia que contenga la cláusula *GROUP BY*, equivale a agrupar todas las filas.

# MD5

```
MD5(string)
```

Calcula un checksum MD5 de 128 bits para la cadena string. El valor se devuelve como un número hexadecimal de 32 dígitos que puede, por ejemplo, usarse como una clave hash:

```
mysql> SELECT MD5("testing");
+-----+
| MD5("testing") |
+-----+
| ae2b1fca515949e5d54fb22b8ed95575 |
+-----+
1 row in set (0.03 sec)
```

Este es el "*RSA Data Security, Inc. MD5 Message-Digest Algorithm*".

# MICROSECOND()

MICROSECOND( expr )

Devuelve los microsegundos a partir de una expresión tiempo o fecha y tiempo como un número en el rango de 0 a 999999.

```
mysql> SELECT MICROSECOND( '12:00:00.123456' );  
      -> 123456  
mysql> SELECT MICROSECOND( '1997-12-31 23:59:59.000010' );  
      -> 10
```

**MICROSECOND()** está disponible desde MySQL 4.1.1.

# SUBSTRING()

## MID()

```

SUBSTRING(cadena, posicion)
SUBSTRING(cadena FROM posicion)
SUBSTRING(cadena, posicion, longitud)
SUBSTRING(cadena FROM posicion FOR longitud)
MID(str, pos, len)

```

Los formatos sin el argumento 'longitud' devuelve una subcadena de la 'cadena' empezando en la 'posicion'. Los formatos con el argumento 'longitud' devuelven una subcadena de 'longitud' caracteres desde la 'cadena', comenzando en la 'posicion'. Los formatos que usan *FROM* tienen la sintaxis **SQL-92**.

```

mysql> SELECT SUBSTRING('MySQL con Clase',7);
+-----+
| SUBSTRING('MySQL con Clase',7) |
+-----+
| con Clase                       |
+-----+
1 row in set (0.00 sec)

mysql> SELECT SUBSTRING('MySQL con Clase' FROM 11);
+-----+
| SUBSTRING('MySQL con Clase' FROM 11) |
+-----+
| Clase                               |
+-----+
1 row in set (0.01 sec)

mysql> SELECT SUBSTRING('MySQL con Clase',7,3);
+-----+
| SUBSTRING('MySQL con Clase',7,3) |
+-----+
| con                               |
+-----+
1 row in set (0.00 sec)

```

Esta función es segura "multi-byte".

**MID(str,pos,len)** es sinónimo de **SUBSTRING(str,pos,len)**.

# MINUTE()

```
MINUTE(time)
```

Devuelve el minuto para el tiempo time, en el rango de 0 a 59:

```
mysql> SELECT MINUTE('98-02-03 10:05:03');
+-----+
| MINUTE('98-02-03 10:05:03') |
+-----+
|                               5 |
+-----+
1 row in set (0.00 sec)
```

# MOD

```
MOD(N,M)  
%
```

Módulo (como el operador % en C). Devuelve el resto de la división de N entre M:

```
mysql> SELECT MOD(234, 10);
```

```
+-----+  
| MOD(234, 10) |  
+-----+  
|           4 |  
+-----+  
1 row in set (0.00 sec)
```

```
mysql> SELECT 253 % 7;
```

```
+-----+  
| 253 % 7 |  
+-----+  
|        1 |  
+-----+  
1 row in set (0.00 sec)
```

```
mysql> SELECT MOD(29,9);
```

```
+-----+  
| MOD(29,9) |  
+-----+  
|          2 |  
+-----+  
1 row in set (0.00 sec)
```

```
mysql> SELECT 29 MOD 9;
```

```
-> 2
```

Esta función es segura con valores *BIGINT*. El último ejemplo sólo funciona en MySQL 4.1.

# MONTH()

```
MONTH(date)
```

Devuelve el mes de una fecha, en el rango de 1 a 12:

```
mysql> SELECT MONTH('1998-02-03');
+-----+
| MONTH('1998-02-03') |
+-----+
|                2 |
+-----+
1 row in set (0.00 sec)
```

# MONTHNAME()

```
MONTHNAME(date)
```

Devuelve el nombre del mes para la fecha date:

```
mysql> SELECT MONTHNAME('1998-02-05');
+-----+
| MONTHNAME('1998-02-05') |
+-----+
| February                 |
+-----+
1 row in set (0.03 sec)
```

# NULLIF

```
NULLIF(expr1, expr2)
```

Devuelve *NULL* si  $\text{expr1} = \text{expr2}$  es verdadero, si no devuelve  $\text{expr1}$ . Esto es lo mismo que si se usa la expresión CASE WHEN  $\text{expr1} = \text{expr2}$  THEN NULL ELSE  $\text{expr1}$  END.

```
mysql> SELECT NULLIF(1,1);  
      -> NULL  
mysql> SELECT NULLIF(1,2);  
      -> 1
```

Hay que tener en cuenta que MySQL evalúa  $\text{expr1}$  dos veces si los argumentos no son iguales. **NULLIF** () se añadió en MySQL 3.23.15.

# OCT()

OCT(N)

Devuelve una cadena que representa el valor octal de N, donde N es un número BIGINT. Es equivalente a [CONV\(N,10,8\)](#). Devuelve NULL si N es NULL:

```
mysql> SELECT OCT(12);
+-----+
| OCT(12) |
+-----+
| 14      |
+-----+
1 row in set (0.00 sec)
```

# PASSWORD

## OLD\_PASSWORD

```
PASSWORD(str)  
OLD_PASSWORD(str)
```

Calcula una cadena contraseña a partir de la cadena en texto plano str. Esta es la función que se usa para encriptar contraseñas MySQL para almacenarlas en las columnas Password de la tabla de concesiones de usuario:

```
mysql> SELECT PASSWORD('badpwd');  
-> '7f84554057dd964b'
```

**PASSWORD()** no es reversible. **PASSWORD()** no realiza una encriptación de la contraseña del mismo modo que se encriptan las contraseñas Unix. Ver [ENCRYPT\(\)](#). Nota: La función **PASSWORD()** se usa para autenticar el sistema en el servidor MySQL, **no se debe usar** en aplicaciones. Para ese propósito usar [MD5\(\)](#) o [SHA1\(\)](#) en su lugar. Ver también *RFC-2195* para más información sobre la manipulación de contraseñas y autenticación segura en aplicaciones.

# ORD()

```
ORD(str)
```

Si el carácter de la izquierda de la cadena str es un carácter multibyte, devuelve el código del carácter, calculado a partir de los valores de los códigos ASCII de los caracteres que lo componen, usando esta fórmula: (código ASCII del primer byte)\*256+(código ASCII del segundo byte)[\*256+código ASCII del tercer byte...]. Si el carácter no es multibyte, devuelve el mismo valor que la función [ASCII\(\)](#):

```
mysql> SELECT ORD('2');
+-----+
| ORD('2') |
+-----+
|          50 |
+-----+
1 row in set (0.00 sec)
```

# PERIOD\_ADD()

```
PERIOD_ADD(P,N)
```

Añade N meses al periodo P (en el formato YYMM o YYYYMM). Devuelve un valor en el formato YYYYMM. El argumento periodo P no es un valor de fecha:

```
mysql> SELECT PERIOD_ADD(9801,2);
+-----+
| PERIOD_ADD(9801,2) |
+-----+
|           199803 |
+-----+
1 row in set (0.00 sec)
```

# PERIOD\_DIFF()

```
PERIOD_DIFF(P1, P2)
```

Devuelve el número de meses entre los periodos P1 y P2. P1 y P2 deben estar en el formato YYMM o YYYYMM. Los argumentos periodo P1 y P2 no son valores de fecha:

```
mysql> SELECT PERIOD_DIFF(9802,199703);
+-----+
| PERIOD_DIFF(9802,199703) |
+-----+
|                          11 |
+-----+
1 row in set (0.02 sec)
```

# PI

```
PI()
```

Devuelve el valor de  $\pi$ . Por defecto se obtienen 5 decimales, pero MySQL usa internamente un valor completo de doble precisión.

```
mysql> SELECT PI();
+-----+
| PI()   |
+-----+
| 3.141593 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT PI()+0.000000000000000000;
+-----+
| PI()+0.000000000000000000 |
+-----+
|      3.141592653589793100 |
+-----+
1 row in set (0.00 sec)
```

# POW

## POWER

```
POW(X,Y)  
POWER(X,Y)
```

Devuelve el valor de X elevado a la potencia Y:

```
mysql> SELECT POW(2,2);  
+-----+  
| POW(2,2) |  
+-----+  
| 4.000000 |  
+-----+  
1 row in set (0.02 sec)  
  
mysql> SELECT POW(2,-2);  
+-----+  
| POW(2,-2) |  
+-----+  
| 0.250000 |  
+-----+  
1 row in set (0.00 sec)
```

# QUARTER()

```
QUARTER(date)
```

Devuelve el cuarto del año para la fecha date, en el rango de 1 a 4:

```
mysql> SELECT QUARTER('98-04-01');
+-----+
| QUARTER('98-04-01') |
+-----+
|                2 |
+-----+
1 row in set (0.00 sec)
```

# QUOTE()

```
QUOTE(str)
```

Entrecomilla una cadena para producir un resultado que se pueda utilizar correctamente como valor escapado en una declaración de los datos SQL. Se devuelve la cadena entre apóstrofes y con cada aparición del carácter '\', ASCII NUL, apóstrofe (") y el Control-Z precedido por un '\'. Si el argumento es NULL, el valor devuelto es la palabra 'NULL' sin las comillas. La función de **QUOTE()** fue agregada en la versión 4.0.3 de MySQL.

```
mysql> SELECT QUOTE("Don't");
```

```
+-----+
| QUOTE("Don't") |
+-----+
| 'Don\'t'       |
+-----+
```

```
1 row in set (0.00 sec)
```

```
mysql> SELECT QUOTE(NULL);
```

```
+-----+
| QUOTE(NULL) |
+-----+
| NULL        |
+-----+
```

```
1 row in set (0.00 sec)
```

# RADIANS

RADIANS(X)

Devuelve el argumento X, convertido de grados a radianes:

```
mysql> SELECT RADIANS(90);
+-----+
| RADIANS(90) |
+-----+
| 1.5707963267949 |
+-----+
1 row in set (0.00 sec)
```

# RAND

```
RAND()  
RAND(N)
```

Devuelve un valor aleatorio en punto flotante, en el rango 0 a 1.0. Si se especifica un argumento entero N, se usa como valor de semilla (produciendo una secuencia repetible):

```
mysql> SELECT RAND();  
+-----+  
| RAND() |  
+-----+  
| 0.40765042361585 |  
+-----+  
1 row in set (0.00 sec)
```

```
mysql> SELECT RAND(20);  
+-----+  
| RAND(20) |  
+-----+  
| 0.15888261251047 |  
+-----+  
1 row in set (0.00 sec)
```

```
mysql> SELECT RAND(20);  
+-----+  
| RAND(20) |  
+-----+  
| 0.15888261251047 |  
+-----+  
1 row in set (0.00 sec)
```

```
mysql> SELECT RAND();  
+-----+  
| RAND() |  
+-----+  
| 0.96999072001315 |  
+-----+  
1 row in set (0.00 sec)
```

```
mysql> SELECT RAND();  
+-----+  
| RAND() |
```

```
+-----+
| 0.62700235995185 |
+-----+
1 row in set (0.00 sec)
```

No es posible usar una columna con valores **RAND()** en una cláusula *ORDER BY*, porque *ORDER BY* puede evaluar la columna varias veces. Desde la versión 3.23 se puede hacer: ***SELECT*** \* ***FROM*** *table\_name* ***ORDER BY*** **RAND()**. Lo siguiente es corriente para obtener una muestra aleatoria de un conjunto ***SELECT*** \* ***FROM*** *table1,table2* ***WHERE*** *a=b* ***AND*** *c<d* ***ORDER BY*** **RAND()** ***LIMIT*** 1000. Hay que tener en cuenta que **RAND()** en una cláusula *WHERE* será evaluada cada vez que el *WHERE* sea ejecutado. **RAND()** no está diseñado para ser un generador aleatorio perfecto, pero sin embargo es un modo rápido de generar números aleatorios que pueden ser portados entre plataformas por la misma versión de MySQL.

# RELEASE\_LOCK

```
RELEASE_LOCK(str)
```

Libera el bloqueo con el nombre `str` que se obtuvo mediante [GET\\_LOCK\(\)](#). Devuelve 1 si el bloqueo fue liberado, 0 si no fue bloqueado por este hilo (en cuyo caso no fue liberado), y NULL si el nombre de bloqueo no existe. (El bloqueo no existirá si nunca fue obtenido por una llamada a [GET\\_LOCK\(\)](#) o si ya ha sido liberado.) Es conveniente usar la sentencia [DO](#) con **RELEASE\_LOCK()**.

# REPEAT()

```
REPEAT(str, count)
```

Devuelve una cadena que consiste en la cadena str repetida count veces. Si count  $\leq 0$ , devuelve una cadena vacía. Devuelve NULL si str o count son NULL:

```
mysql> SELECT REPEAT('MySQL', 3);
+-----+
| REPEAT('MySQL', 3) |
+-----+
| MySQLMySQLMySQL    |
+-----+
1 row in set (0.00 sec)
```

# REPLACE()

```
REPLACE(str,from_str,to_str)
```

Devuelve la cadena str con todas las apariciones de la cadena from\_str sustituidas por la cadena to\_str:

```
mysql> SELECT REPLACE('www.mysql.com', 'w', 'Ww');
+-----+
| REPLACE('www.mysql.com', 'w', 'Ww') |
+-----+
| WwWwWw.mysql.com                    |
+-----+
1 row in set (0.00 sec)
```

Esta función es segura a para caracteres multibyte.

# REVERSE()

```
REVERSE(str)
```

Devuelve la cadena str con el orden de los caracteres invertido:

```
mysql> SELECT REVERSE('abc');  
      -> 'cba'
```

Esta función es segura con caracteres multibyte.

# RIGHT()

```
RIGHT(cadena, longitud)
```

Devuelve los 'longitud' caracteres de la derecha de la 'cadena':

```
mysql> SELECT RIGHT('MySQL con Clase', 9);
+-----+
| RIGHT('MySQL con Clase', 9) |
+-----+
| con Clase                    |
+-----+
1 row in set (0.00 sec)
```

Esta función es segura "multi-byte".

# ROUND

```
ROUND(X)
ROUND(X,D)
```

Devuelve el argumento X, redondeado al entero más cercano. Con dos argumentos redondea a un número con D decimales.

```
mysql> SELECT ROUND(-1.23);
+-----+
| ROUND(-1.23) |
+-----+
|           -1 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT ROUND(-1.58);
+-----+
| ROUND(-1.58) |
+-----+
|           -2 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT ROUND(1.58);
+-----+
| ROUND(1.58) |
+-----+
|           2 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT ROUND(1.298, 1);
+-----+
| ROUND(1.298, 1) |
+-----+
|           1.3 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT ROUND(1.298, 0);
+-----+
| ROUND(1.298, 0) |
```

## ROUND

```
+-----+
|          1 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT ROUND(23.298, -1);
+-----+
| ROUND(23.298, -1) |
+-----+
|          20 |
+-----+
1 row in set (0.00 sec)
```

El comportamiento de **ROUND()** cuando el argumento está justo en la mitad de dos enteros depende de la implementación de la librería C. Algunos redondean al número impar más cercano, otros hacia arriba, hacia abajo, o hacia cero. Si se necesita un tipo de redondeo, se puede usar la función [TRUNCATE\(\)](#) o [FLOOR\(\)](#) en su lugar.

# RPAD()

```
RPAD(str,len,padstr)
```

Devuelve la cadena str, rellena a la derecha con la cadena padstr hasta la longitud de len caracteres. Si str es más larga que len, el valor retornado se acorta hasta len caracteres.

```
mysql> SELECT RPAD('hi',5,'?');
+-----+
| RPAD('hi',5,'?') |
+-----+
| hi???           |
+-----+
1 row in set (0.00 sec)
```

Esta función es segura con caracteres multibyte.

# RTRIM()

```
RTRIM(str)
```

Devuelve la cadena str con los caracteres de espacios finales eliminados:

```
mysql> SELECT RTRIM('barbar  ');  
      -> 'barbar'
```

Esta función es segura con caracteres multibyte.

# SECOND()

```
SECOND(time)
```

Devuelve el segundo para el tiempo time, en el rango de 0 a 59:

```
mysql> SELECT SECOND('10:05:03');
+-----+
| SECOND('10:05:03') |
+-----+
|                    3 |
+-----+
1 row in set (0.00 sec)
```

# SEC\_TO\_TIME()

```
SEC_TO_TIME(seconds)
```

Devuelve el argumento seconds, convertido a horas, minutos y segundos, como un valor en el formato 'HH:MM:SS' o HHMMSS, dependiendo de si la función se usa en un contexto de cadena o numérico:

```
mysql> SELECT SEC_TO_TIME(2378);
+-----+
| SEC_TO_TIME(2378) |
+-----+
| 00:39:38          |
+-----+
1 row in set (0.00 sec)

mysql> SELECT SEC_TO_TIME(2378) + 0;
+-----+
| SEC_TO_TIME(2378) + 0 |
+-----+
|                3938 |
+-----+
1 row in set (0.00 sec)
```

# **USER**

## **SESSION\_USER**

## **SYSTEM\_USER**

```
USER()  
SESSION_USER()  
SYSTEM_USER()
```

Devuelve el nombre de usuario y host actual de MySQL:

```
mysql> SELECT USER();  
      -> 'davida@localhost'
```

El valor indica en nombre de usuario que se especificó cuando se conectó al servidor, y el host cliente desde el que se conectó. (En versiones anteriores a MySQL 3.22.11, el valor de la función no incluye el nombre del host del cliente.) Se puede extraer sólo la parte del nombre de usuario, sin tener en cuenta si se incluye o no la parte del nombre del host de esta forma:

```
mysql> SELECT SUBSTRING_INDEX(USER(), "@", 1);  
      -> 'davida'
```

**SYSTEM\_USER()** y **SESSION\_USER()** son sinónimos de **USER()**.

# SHA

## SHA1

```
SHA1(string)  
SHA(string)
```

Calcula un checksum SHA1 de 160 bits para la cadena string, como se describe en RFC 3174 (Secure Hash Algorithm). El valor se devuelve como un número hexadecimal de 40 dígitos, o NULL en caso de que el argumento de entrada sea NULL. Uno de los posibles usos de esta función es como una clave hash. También se puede usar como una función de seguridad criptográfica para almacenar contraseñas.

```
mysql> SELECT SHA1("abc");  
-> 'a9993e364706816aba3e25717850c26c9cd0d89d'
```

**SHA1()** se añadió en la versión 4.0.2, y puede considerarse un equivalente de [MD5\(\)](#) más seguro criptográficamente. **SHA()** es sinónimo de **SHA1()**.

# ***SIGN***

SIGN(X)

Devuelve el signo del argumento como -1, 0 ó 1, dependiendo de si X es negativo, cero o positivo:

```
mysql> SELECT SIGN(-32);
+-----+
| SIGN(-32) |
+-----+
|          -1 |
+-----+
1 row in set (0.00 sec)
```

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

```
mysql> SELECT SIGN(234);
+-----+
| SIGN(234) |
+-----+
|          1 |
+-----+
1 row in set (0.00 sec)
```

# SIN

```
SIN(X)
```

Devuelve el seno de X, donde X viene dado en radianes:

```
mysql> SELECT SIN(PI());  
+-----+  
| SIN(PI()) |  
+-----+  
| 0.000000 |  
+-----+  
1 row in set (0.00 sec)
```

# SOUNDEX()

```
SOUNDEX(str)
```

Devuelve una cadena 'soundex' desde str. Dos cadenas que suenen casi igual tienen cadenas soundex idénticas. Una cade soundex estándar tiene 4 caracteres de longitud, pero **SOUNDEX()** devuelve una cadena de longitud arbitraria. Se puede usar **SUBSTRING()** sobre el resultado para obtener una cadena soundex estándar. Cualquier carácter no alfanumérico será ignorado. Todos los caracteres alfabéticos fuera del rango A-Z serán tratados como vocales:

```
mysql> SELECT SOUNDEX('Hello');
+-----+
| SOUNDEX('Hello') |
+-----+
| H400              |
+-----+
1 row in set (0.00 sec)

mysql> SELECT SOUNDEX('Quadratically');
+-----+
| SOUNDEX('Quadratically') |
+-----+
| Q36324                  |
+-----+
1 row in set (0.00 sec)
```

```
expr1 SOUNDS LIKE expr2
```

Es equivalente a:

```
SOUNDEX(expr1)=SOUNDEX(expr2)
```

Disponible sólo a partir de la versión 4.1.

# SOUNDS LIKE

```
expr1 SOUNDS LIKE expr2
```

Es lo mismo que hacer la comparación SOUNDEX(*expr1*) = *SOUNDEX*(*expr2*). Está disponible sólo a partir de MySQL 4.1.

# SPACE()

SPACE(N)

Devuelve una cadena que consiste en N caracteres espacio:

```
mysql> SELECT SPACE(6);
+-----+
| SPACE(6) |
+-----+
|          |
+-----+
1 row in set (0.00 sec)
```

# SQRT

SQRT(X)

Devuelve la raíz cuadrada no negativa de X:

```
mysql> SELECT SQRT(4);
+-----+
| SQRT(4) |
+-----+
| 2.000000 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT SQRT(20);
+-----+
| SQRT(20) |
+-----+
| 4.472136 |
+-----+
1 row in set (0.00 sec)
```

# STD

## STDDEV

```
STD ( expr )  
STDDEV ( expr )
```

Devuelve la desviación estándar de la expresión (la raíz cuadrada de [VARIANCE\(\)](#)). Esta es una extensión para SQL-99. El formato de **STDDEV()** de esta función se proporciona para compatibilidad con **Oracle**.

Si se usa una función de grupo en una sentencia que contenga la cláusula *GROUP BY*, equivale a agrupar todas las filas.

# STRCMP

```
STRCMP(expr1,expr2)
```

**STRCMP()** devuelve 0 si las cadenas son iguales, -1 si el primer argumento es menor que el segundo, según el orden de ordenamiento de cadenas actual, y 1 en otro caso.

```
mysql> SELECT STRCMP('text', 'text2');
      -> -1
mysql> SELECT STRCMP('text2', 'text');
      -> 1
mysql> SELECT STRCMP('text', 'text');
      -> 0
```

Desde MySQL 4.0, **STRCMP()** usa el conjunto de caracteres actual cuando realiza las comparaciones. Esto hace que el comportamiento de comparación por defecto no sea sensible al tipo, a no ser que uno o ambos operandos sean cadenas binarias. Antes de MySQL 4.0, **STRCMP()** es sensible al tipo.

# STR\_TO\_DATE()

```
STR_TO_DATE(str,format)
```

Esta es la función inversa de la función [DATE\\_FORMAT\(\)](#). Toma una cadena str, y una cadena format, y devuelve un valor *DATETIME*. Los valores date, time o datetime contenidos en str deben ser dados en el formato indicado por format. Para ver los especificadores que pueden ser usados en format, ver la tabla en la descripción de la función [DATE\\_FORMAT\(\)](#). El resto de los caracteres se toman tal cual, y no son interpretadas. Si str contiene una fecha, un valor tiempo o de fecha y tiempo ilegal, **STR\_TO\_DATE()** devuelve NULL.

```
mysql> SELECT STR_TO_DATE('03.10.2003 09.20', '%d.%m.%Y %H.%i');
      -> 2003-10-03 09:20:00
mysql> SELECT STR_TO_DATE('10rap', '%crap');
      -> 0000-10-00 00:00:00
mysql> SELECT STR_TO_DATE('2003-15-10 00:00:00', '%Y-%m-%d %H:%i:%s');
      -> NULL
```

**STR\_TO\_DATE()** está disponible desde MySQL 4.1.1.

# SUBDATE()

```
SUBDATE(date, INTERVAL expr type)
SUBDATE(expr, days)
```

Cuando se invoca con el formato *INTERVAL* en el segundo argumento, **SUBDATE()** es un sinónimo de [DATE\\_SUB\(\)](#).

```
mysql> SELECT DATE_SUB('1998-01-02', INTERVAL 31 DAY);
+-----+
| DATE_SUB('1998-01-02', INTERVAL 31 DAY) |
+-----+
| 1997-12-02                               |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT SUBDATE('1998-01-02', INTERVAL 31 DAY);
+-----+
| SUBDATE('1998-01-02', INTERVAL 31 DAY) |
+-----+
| 1997-12-02                               |
+-----+
1 row in set (0.00 sec)
```

Desde la versión MySQL 4.1.1, se permite la segunda sintaxis, donde *expr* es una expresión de fecha o de fecha y hora y *days* es el número de días a restar desde *expr*.

```
mysql> SELECT SUBDATE('1998-01-02 12:00:00', 31);
-> '1997-12-02 12:00:00'
```

# SUBSTRING\_INDEX()

```
SUBSTRING_INDEX(str,delim,count)
```

Devuelve la subcadena de str anterior a la aparición de count veces el delimitador delim. Si count es positivo, se retorna todo lo que haya a la izquierda del delimitador final (contando desde la izquierda). Si count es negativo, se devuelve todo lo que haya a la derecha del delimitador final (contando desde la derecha):

```
mysql> SELECT SUBSTRING_INDEX('www.mysql.com', '.', 2);
```

```
+-----+
| SUBSTRING_INDEX('www.mysql.com', '.', 2) |
+-----+
| www.mysql                               |
+-----+
```

```
1 row in set (0.00 sec)
```

```
mysql> SELECT SUBSTRING_INDEX('www.mysql.com', '.', -2);
```

```
+-----+
| SUBSTRING_INDEX('www.mysql.com', '.', -2) |
+-----+
| mysql.com                                 |
+-----+
```

```
1 row in set (0.00 sec)
```

Esta función es segura "multi-byte".

# SUBTIME()

```
SUBTIME(expr, expr2)
```

**SUBTIME()** resta `expr2` de `expr` y devuelve el resultado. `expr` es una expresión fecha o fecha y hora, y `expr2` es una expresión tiempo.

```
mysql> SELECT SUBTIME("1997-12-31 23:59:59.999999", "1 1:1:1.000002");
      -> '1997-12-30 22:58:58.999997'
mysql> SELECT SUBTIME("01:00:00.999999", "02:00:00.999998");
      -> '-00:59:59.999999'
```

**SUBTIME()** se añadió en MySQL 4.1.1.

# SUM

```
SUM(expr)
```

Devuelve la suma de la expresión *expr*. Si el conjunto de resultados no tiene filas, devuelve NULL.

Si se usa una función de grupo en una sentencia que contenga la cláusula *GROUP BY*, equivale a agrupar todas las filas.

# TAN

TAN(X)

Devuelve la tangente de X, donde X viene dado en radianes:

```
mysql> SELECT TAN(PI()+1);
+-----+
| TAN(PI()+1) |
+-----+
|      1.557408 |
+-----+
1 row in set (0.00 sec)
```

# TIME()

```
TIME( expr )
```

Extrae la parte de la hora de la expresión expr del tipo tiempo o fecha y hora.

```
mysql> SELECT TIME('2003-12-31 01:02:03');  
      -> '01:02:03'  
mysql> SELECT TIME('2003-12-31 01:02:03.000123');  
      -> '01:02:03.000123'
```

**TIME()** se añadió en MySQL 4.1.1.

# TIMEDIFF()

```
TIMEDIFF( expr , expr2 )
```

**TIMEDIFF()** devuelve el tiempo entre la expresión de tiempo de inicio `expr` y la final `expr2`. `expr` y `expr2` son expresiones tiempo de fecha y hora, pero ambas deben ser del mismo tipo.

```
mysql> SELECT TIMEDIFF('2000:01:01 00:00:00', '2000:01:01
00:00:00.000001');
      -> '-00:00:00.000001'
mysql> SELECT TIMEDIFF('1997-12-31 23:59:59.000001', '1997-12-30
01:01:01.000002');
      -> '46:58:57.999999'
```

**TIMEDIFF()** se añadió en MySQL 4.1.1.

# TIMESTAMP()

```
TIMESTAMP(expr)
TIMESTAMP(expr,expr2)
```

Con un argumento, devuelve la expresión `expr` de fecha o fecha y hora como un valor fecha y tiempo.  
Con dos argumentos, suma la expresión de tiempo `expr2` a la expresión de fecha o fecha y hora `expr` y devuelve un valor de fecha y tiempo.

```
mysql> SELECT TIMESTAMP('2003-12-31');
      -> '2003-12-31 00:00:00'
mysql> SELECT TIMESTAMP('2003-12-31 12:00:00','12:00:00');
      -> '2004-01-01 00:00:00'
```

**TIMESTAMP()** se añadió en MySQL 4.1.1.

# TIMESTAMPADD()

```
TIMESTAMPADD(interval,int_expr,datetime_expr)
```

Suma la expresión entera `int_expr` a la expresión de fecha o fecha y hora `datetime_expr`. Las unidades para `int_expr` se toman del argumento `interval`, que debe ser uno de los siguiente: *FRAC\_SECOND*, *SECOND*, *MINUTE*, *HOURL*, *DAY*, *WEEK*, *MONTH*, *QUARTER* o *YEAR*. El valor de `interval` puede ser especificado usando una de las palabras clave mostradas, o con un prefijo de `SQL_TSI_`. Por ejemplo, *DAY* o *SQL\_TSI\_DAY* son ambas legales.

```
mysql> SELECT TIMESTAMPADD(MINUTE,1,'2003-01-02');
-> '2003-01-02 00:01:00'
mysql> SELECT TIMESTAMPADD(WEEK,1,'2003-01-02');
-> '2003-01-09'
```

**TIMESTAMPADD()** se añadió en MySQL 5.0.0.

# TIMESTAMPDIFF()

```
TIMESTAMPDIFF(interval,datetime_expr1,datetime_expr2)
```

Devuelve la diferencia entera entre las expresiones fecha o fecha y hora `datetime_expr1` y `datetime_expr2`. Las unidades para el resultado vienen dadas por el argumento `interval`. Los valores legales para `interval` son los mismos que se mencionan en la descripción de la función [TIMESTAMPADD\(\)](#).

```
mysql> SELECT TIMESTAMPDIFF(MONTH, '2003-02-01', '2003-05-01');
-> 3
mysql> SELECT TIMESTAMPDIFF(YEAR, '2002-05-01', '2001-01-01');
-> -1
```

**TIMESTAMPDIFF()** se añadió en MySQL 5.0.0.

# TIME\_FORMAT()

```
TIME_FORMAT(time,format)
```

Se usa como la función [DATE\\_FORMAT\(\)](#), pero la cadena de formato sólo puede contener aquellos especificadores de formato que manejan horas, minutos y segundos. Otros especificadores producen un valor NULL o 0. Si el valor de tiempo contiene una parte de hora mayor que 23, los especificadores de formato de hora %H y %k producen un valor mayor que el usual de 0..23. Los otros especificadores de formato de hora producen el valor de hora módulo 12:

```
mysql> SELECT TIME_FORMAT('100:00:00', '%H %k %h %I %l');
+-----+
| TIME_FORMAT('100:00:00', '%H %k %h %I %l') |
+-----+
| 100 100 04 04 4 |
+-----+
1 row in set (0.02 sec)
```

# TIME\_TO\_SEC()

```
TIME_TO_SEC(time)
```

Devuelve el argumento time convertido en segundos:

```
mysql> SELECT TIME_TO_SEC('22:23:00');
+-----+
| TIME_TO_SEC('22:23:00') |
+-----+
|                80580 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT TIME_TO_SEC('00:39:38');
+-----+
| TIME_TO_SEC('00:39:38') |
+-----+
|                2378 |
+-----+
1 row in set (0.00 sec)
```

# TO\_DAYS()

```
TO_DAYS(date)
```

Dada la fecha date, devuelve el número del día (el número de días desde el año 0):

```
mysql> SELECT TO_DAYS(950501);
```

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

```
| TO_DAYS(950501) |
```

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

```
|           728779 |
```

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

```
1 row in set (0.00 sec)
```

```
mysql> SELECT TO_DAYS('1997-10-07');
```

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

```
| TO_DAYS('1997-10-07') |
```

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

```
|           729669 |
```

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

```
1 row in set (0.00 sec)
```

**TO\_DAYS()** no está diseñada para trabajar con valores anteriores a la implantación del calendario Gregoriano (1582), porque no tiene en cuenta los días perdidos cuando se instauró dicho calendario.

# TRIM

```
TRIM([[BOTH | LEADING | TRAILING] [remstr] FROM] str)
```

Devuelve la cadena `str` eliminando todos los prefijos y/o sufijos `remstr`. Si no se incluye ninguno de los especificadores *BOTH*, *LEADING* o *TRAILING*, se asume *BOTH*. Si no se especifica la cadena `remstr`, se eliminan los espacios:

```
mysql> SELECT TRIM('  bar  ');
+-----+
| TRIM(LEADING 'x' FROM 'xxxbarxxx') |
+-----+
| barxxx                               |
+-----+
1 row in set (0.00 sec)

mysql> SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx');
+-----+
| TRIM(BOTH 'x' FROM 'xxxbarxxx') |
+-----+
| bar                               |
+-----+
1 row in set (0.00 sec)

mysql> SELECT TRIM(TRAILING 'xyz' FROM 'barxyz');
+-----+
| TRIM(TRAILING 'xyz' FROM 'barxyz') |
+-----+
| barx                               |
+-----+
1 row in set (0.00 sec)
```

Esta función es segura "multi-byte".

# TRUNCATE

```
TRUNCATE(X,D)
```

Devuelve el número X, truncado a D decimales. Si D es 0, el resultado no tendrá punto decimal o parte fraccionaria:

```
mysql> SELECT TRUNCATE(1.223,1);
```

```
+-----+
| TRUNCATE(1.223,1) |
+-----+
|                1.2 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT TRUNCATE(1.999,1);
```

```
+-----+
| TRUNCATE(1.999,1) |
+-----+
|                1.9 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT TRUNCATE(1.999,0);
```

```
+-----+
| TRUNCATE(1.999,0) |
+-----+
|                1 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT TRUNCATE(-1.999,1);
```

```
+-----+
| TRUNCATE(-1.999,1) |
+-----+
|               -1.9 |
+-----+
1 row in set (0.00 sec)
```

Desde MySQL 3.23.51, todos los números se redondean hacia cero. Si D es negativo, entonces la parte correspondiente del número es asignada a cero:

```
mysql> SELECT TRUNCATE(122,-2);
+-----+
| TRUNCATE(122,-2) |
+-----+
|                100 |
+-----+
1 row in set (0.00 sec)
```

Como los números decimales normalmente no se almacenan como números exactos en los ordenadores, sino como valores de doble precisión, el siguiente resultado puede parecer sorprendente:

```
mysql> SELECT TRUNCATE(10.28*100,0);
+-----+
| TRUNCATE(10.28*100,0) |
+-----+
|                1027 |
+-----+
1 row in set (0.00 sec)
```

Lo anterior sucede porque 10.28 se almacena como algo parecido a 10.2799999999999999.

# UCASE()

## UPPER()

```
UCASE(str)  
UPPER(str)
```

Devuelve la cadena `str` con todos sus caracteres sustituidos a mayúsculas de acuerdo con el mapa del conjunto de caracteres actual (por defecto es ISO-8859-1 Latin1):

```
mysql> SELECT UPPER('Hej');  
      -> 'HEJ'
```

Esta función es segura "multi-byte".

**UCASE()** es sinónimo de **UPPER()**.

# UNCOMPRESS

```
UNCOMPRESS(string_to_uncompress)
```

Descomprime una cadena comprimida con la función [COMPRESS\(\)](#).

```
mysql> SELECT UNCOMPRESS(COMPRESS("any string"));  
      -> 'any string'
```

**UNCOMPRESS()** se añadió en MySQL 4.1.1. Requiere que MySQL haya sido compilado con una librería de compresión como **zlib**. De otro modo, el valor de retorno es siempre NULL.

# UNCOMPRESSED\_LENGTH

```
UNCOMPRESSED_LENGTH(compressed_string)
```

Devuelve la longitud de una cadena comprimida antes de su compresión.

```
mysql> SELECT UNCOMPRESSED_LENGTH(COMPRESS(REPEAT("a",30)));  
      -> 30
```

**UNCOMPRESSED\_LENGTH()** se añadió en MySQL 4.1.1.

# UNHEX

```
UNHEX(str)
```

Es la función opuesta a [HEX\(str\)](#). Es decir, interpreta cada par de dígitos hexadecimales del argumento como un número, y lo convierte en el carácter representado por ese número. Los caracteres resultantes se devuelven como una cadena binaria.

```
mysql> SELECT UNHEX('4D7953514C');  
      -> 'MySQL'  
mysql> SELECT 0x4D7953514C;  
      -> 'MySQL'  
mysql> SELECT UNHEX(HEX('string'));  
      -> 'string'  
mysql> SELECT HEX(UNHEX('1267'));  
      -> '1267'
```

**UNHEX()** fue añadido en MySQL 4.1.2.

# UNIX\_TIMESTAMP()

```
UNIX_TIMESTAMP()
UNIX_TIMESTAMP(date)
```

Si es llamada sin argumentos, devuelve un timestamp Unix (segundos desde '1970-01-01 00:00:00' GMT) como un entero sin signo. Si es llamada con un argumento fecha, devuelve el valor del argumento como segundos desde '1970-01-01 00:00:00' GMT. *date* debe ser una cadena *DATE*, una cadena *DATETIME*, un *TIMESTAMP* o un número en el formato *YYMMDD* o *YYYYMMDD* en hora local:

```
mysql> SELECT UNIX_TIMESTAMP();
+-----+
| UNIX_TIMESTAMP() |
+-----+
|          107255517 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT UNIX_TIMESTAMP('1997-10-04 22:23:00');
+-----+
| UNIX_TIMESTAMP('1997-10-04 22:23:00') |
+-----+
|                                875996580 |
+-----+
1 row in set (0.02 sec)
```

Cuando se usa **UNIX\_TIMESTAMP** en una columna *TIMESTAMP*, la función devuelve el valor interno timestamp directamente, sin la conversión implícita "string-to-Unix-timestamp". Si se usa una fecha fuera de rango a **UNIX\_TIMESTAMP()** devuelve 0, pero hay que tener en cuenta que sólo se hace una comprobación básica (año en el margen 1970-2037, mes en 01-12 y día en 01-31). Si se quieren restar columnas **UNIX\_TIMESTAMP()**, **puede desearse convertir el resultado a enteros con signo.**

# ***UTC\_DATE***

## ***UTC\_DATE()***

```
UTC_DATE  
UTC_DATE()
```

Devuelve la fecha UTC actual como un valor en el formato 'YYYY-MM-DD' o YYYYMMDD, dependiendo de si se usa en un contexto de cadena o numérico:

```
mysql> SELECT UTC_DATE(), UTC_DATE() + 0;  
      -> '2003-08-14', 20030814
```

**UTC\_DATE()** está disponible desde MySQL 4.1.1.

# ***UTC\_TIME***

## ***UTC\_TIME()***

```
UTC_TIME  
UTC_TIME()
```

Devuelve la hora UTC actual como un valor en el formato 'HH:MM:SS' o HHMMSS, dependiendo de si se usa en un contexto de cadena o numérico:

```
mysql> SELECT UTC_TIME(), UTC_TIME() + 0;  
      -> '18:07:53', 180753
```

**UTC\_TIME()** está disponible desde MySQL 4.1.1.

# ***UTC\_TIMESTAMP***

## ***UTC\_TIMESTAMP()***

```
UTC_TIMESTAMP  
UTC_TIMESTAMP()
```

Devuelve la fecha y hora UTC actual como un valor en el formato 'YYYY-MM-DD HH:MM:SS' o YYYYMMDDHHMMSS, dependiendo de si se usa en un contexto de cadena o numérico:

```
mysql> SELECT UTC_TIMESTAMP(), UTC_TIMESTAMP() + 0;  
-> '2003-08-14 18:08:04', 20030814180804
```

**UTC\_TIMESTAMP()** está disponible desde MySQL 4.1.1.

# UUID

```
UUID( )
```

Devuelve un identificador único universal (UUID) generado de acuerdo con el "Procedimiento de llamada remota DCE 1.1" (Apéndice A) Especificaciones de CAE (Common Applications Environment), publicadas por "The Open Group" en Octubre de 1997 (Documento número C706). Un UUID está diseñado como un número que es único globalmente en el espacio y el tiempo. Es de esperar que dos llamadas a **UUID()** generen dos valores diferentes, aunque esas llamadas se realicen en dos ordenadores separados que no estén conectados entre ellos. Un UUID is es un número de 128 bits representado por una cadena de cinco números hexadecimales en el formato aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee format:

- Los tres primeros números se generan a partir de un timestamp.
- El cuarto número preserva la unicidad temporal en el caso en que el valor de timestamp pierda la secuencia (por ejemplo, en caso de cambio de hora para ahorro de luz diurna).
- El quinto número es un número de nodo IEEE 802 que proporciona una unicidad espacial. Se sustituye por un número aleatorio si este último no está disponible (por ejemplo, porque el ordenador no dispone de una tarjeta Ethernet, o no se sabe cómo obtener la dirección de hardware del interfaz en el sistema operativo). En este caso, la unicidad espacial no puede ser garantizada. Sin embargo, una colisión tiene muy pocas probabilidades. Actualmente, la dirección MAC de un interfaz se tiene en cuenta sólo en FreeBSD y Linux. En otros sistemas operativos, MySQL usa un número aleatorio genrado de 48 bits.

```
mysql> SELECT UUID();  
-> '6ccd780c-baba-1026-9564-0040f4311e29'
```

Hay que tener en cuenta que **UUID()** aún no trabaja con replica. **UUID()** se añadió en MySQL 4.1.2.

# VARIANCE

```
VARIANCE ( expr )
```

Devuelve la varianza estándar de la expresión `expr` (condiderando la filas como la población completa, no como una muestra; de modo que usa el número de filas como denominador). Esto es una extensión a SQL-99 (disponible sólo en versión 4.1 o superior).

Si se usa una función de grupo en una sentencia que contenga la cláusula *GROUP BY*, equivale a agrupar todas las filas.

# VERSION()

```
VERSION()
```

Devuelve una cadena que indica la versión del servidor MySQL:

```
mysql> SELECT VERSION();
+-----+
| VERSION() |
+-----+
| 4.0.15-nt |
+-----+
1 row in set (0.00 sec)
```

Si la versión termina con -log significa que está activado el diario (log).

# WEEK()

```
WEEK(date [,mode])
```

Esta función devuelve el número de la semana para una fecha. El formato con dos argumentos permite especificar si la semana empieza en domingo o en lunes y si el valor de retorno debe estar en el rango 0-53 o 1-52. Cuando se omite el argumento de modo el valor por defecto usado es el de la variable del servidor `default_week_format` (o 0 en MySQL 4.0 o anterior). La tabla siguiente demuestra cómo trabaja el argumento `mode`:

Valor	Significado
0	La semana empieza en domingo; devuelve un valor en el rango 0 a 53; la semana 1 es la primera semana que empieza en este año
1	La semana empieza en lunes; devuelve un valor en el rango 0 a 53; la semana 1 es la primera semana que tiene más de 3 días en este año
2	La semana empieza en domingo; devuelve un valor en el rango 1 a 53; la semana 1 es la primera semana que empieza en este año
3	La semana empieza en lunes; devuelve un valor en el rango 1 a 53; la semana 1 es la primera semana que tenga más de tres días en este año
4	La semana empieza en domingo; devuelve un valor en el rango 0 a 53; la semana 1 es la primera semana que tenga más de tres días en este año
5	La semana empieza en lunes; devuelve un valor en el rango 0 a 53; la semana 1 es la primera semana que empiece en este año
6	La semana empieza en domingo; devuelve un valor en el rango 1 a 53; la semana 1 es la primera semana que tenga más de tres días en este año
7	La semana empieza en lunes; devuelve un valor en el rango 1 a 53; la semana 1 es la primera semana que empiece en este año

El valor de modo 3 puede usarse desde MySQL 4.0.5. El valor de modo 4 y superiores pueden usarse desde MySQL 4.0.17.

```
mysql> SELECT WEEK('1998-02-20');
+-----+
| WEEK('1998-02-20') |
+-----+
|                    7 |
```

## WEEK

```
+-----+
1 row in set (0.00 sec)

mysql> SELECT WEEK('1998-02-20',0);
+-----+
| WEEK('1998-02-20',0) |
+-----+
|                7 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT WEEK('1998-02-20',1);
+-----+
| WEEK('1998-02-20',1) |
+-----+
|                8 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT WEEK('1998-12-31',1);
+-----+
| WEEK('1998-12-31',1) |
+-----+
|                53 |
+-----+
1 row in set (0.00 sec)
```

Nota: en la versión 4.0, **WEEK(date,0)** se modificó para que coincidiera con el calendario en USA. Antes de eso, **WEEK()** calculaba de forma incorrecta para fechas en USA. (En efecto, **WEEK(date)** y **WEEK(date,0)** era incorrecto en todos los casos.) So una fecha cae en la última semana del año anterior, MySQL retornará 0 si no se usa 2, 3, 6 o 7 como valor del argumento opcional mode:

```
mysql> SELECT YEAR('2000-01-01'), WEEK('2000-01-01',0);
+-----+-----+
| YEAR('2000-01-01') | WEEK('2000-01-01',0) |
+-----+-----+
|                2000 |                0 |
+-----+-----+
1 row in set (0.00 sec)
```

Se puede argumentar que MySQL debe devolver 52 en la función **WEEK()**, porque la fecha dada está en la semana 53 de 1999. Pero se ha decidido devolver 0 en su lugar ya que se prefiere que la función devuelva "el número de la semana en el año dado". Esto hace el uso de la función **WEEK()** función más fiable cuando se combina con otras funciones que extraen una parte de la fecha. Si se prefiere que el

resultado sea evaluado con respecto al año que contiene el primer día de la semana para la fecha dada, se debe usar 2, 3, 6 ó 7 en el argumento mode.

```
mysql> SELECT WEEK('2000-01-01',2);
+-----+
| WEEK('2000-01-01',2) |
+-----+
|                    52 |
+-----+
1 row in set (0.00 sec)
```

Alternativamente, se puede usar la función [YEARWEEK\(\)](#):

```
mysql> SELECT YEARWEEK('2000-01-01');
+-----+
| YEARWEEK('2000-01-01') |
+-----+
|                199952 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT MID(YEARWEEK('2000-01-01'),5,2);
+-----+
| MID(YEARWEEK('2000-01-01'),5,2) |
+-----+
| 52                               |
+-----+
1 row in set (0.00 sec)
```

# WEEKDAY()

```
WEEKDAY(date)
```

Devuelve el índice del día de la semana para la fecha date (0 = Lunes, 1 = Martes, ... 6 = Domingo):

```
mysql> SELECT WEEKDAY('1998-02-03 22:23:00');
```

```
+-----+
| WEEKDAY('1998-02-03 22:23:00') |
+-----+
|                               1 |
+-----+
```

```
1 row in set (0.00 sec)
```

```
mysql> SELECT WEEKDAY('1997-11-05');
```

```
+-----+
| WEEKDAY('1997-11-05') |
+-----+
|                       2 |
+-----+
```

```
1 row in set (0.00 sec)
```

# WEEKOFYEAR()

```
WEEKOFYEAR(date)
```

Devuelve el número de semana según el calendario de la fecha dada como un número en el rango de 1 a 53.

```
mysql> SELECT WEEKOFYEAR('1998-02-20');  
-> 8
```

**WEEKOFYEAR()** está disponible desde MySQL 4.1.1.

# YEAR()

```
YEAR(date)
```

Devuelve el año para una fecha, en el rango de 1000 a 9999:

```
mysql> SELECT YEAR('98-02-03');
```

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

```
| YEAR('98-02-03') |
```

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

```
|           1998 |
```

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

```
1 row in set (0.00 sec)
```

# YEARWEEK()

```
YEARWEEK(date)
YEARWEEK(date,start)
```

Devuelve el año y semana de una fecha. El argumento start trabaja exactamente igual que el argumento argument en la función [WEEK\(\)](#). El año en el resultado puede ser diferente del año en el argumento date para la primera y última semana del año:

```
mysql> SELECT YEARWEEK('1987-01-01');
+-----+
| YEARWEEK('1987-01-01') |
+-----+
|           198652      |
+-----+
1 row in set (0.00 sec)
```

El número de semana es diferente que para la función [WEEK\(\)](#), que puede devolver el valor 0 para los argumentos opcionales 0 o 1, ya que [WEEK\(\)](#) en ese caso, devuelve la semana en el contexto del año dado.

# Indice de Funciones y tipos API C SQL (72)

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

En mayúsculas aparecen los tipos definidos en **MySQL**, en minúsculas, las funciones del API C.

-\_-



MYSQL

-A-



mysql\_affected\_rows

mysql\_autocommit

-C-



mysql\_change\_user

mysql\_character\_set\_name

mysql\_close

mysql\_commit

mysql\_connect

mysql\_create\_db

-D-



MYSQL DATA

mysql\_data\_seek

mysql\_debug

mysql\_drop\_db

mysql\_dump\_debug\_info

-E-



mysql\_eof

mysql\_errno

mysql\_error

mysql\_escape\_string

-F-



mysql\_fetch\_field

mysql\_fetch\_fields

mysql\_fetch\_field\_direct

mysql\_fetch\_lengths

mysql\_fetch\_row

MYSQL\_FIELD

mysql\_field\_count

mysql\_field\_seek

mysql\_field\_tell

mysql\_free\_result

-G-



mysql\_get\_client\_info

mysql\_get\_client\_version

[mysql\\_get\\_host\\_info](#)

[mysql\\_get\\_proto\\_info](#)

[mysql\\_get\\_server\\_info](#)

[mysql\\_get\\_server\\_version](#)

-H- 

[mysql\\_hex\\_string](#)

-I- 

[mysql\\_info](#)

[mysql\\_init](#)

[mysql\\_insert\\_id](#)

-K- 

[mysql\\_kill](#)

-L- 

[mysql\\_library\\_end](#)

[mysql\\_library\\_init](#)

[mysql\\_list\\_dbs](#)

[mysql\\_list\\_fields](#)

[mysql\\_list\\_processes](#)

[mysql\\_list\\_tables](#)

-M- 

[mysql\\_more\\_results](#)

-N- 

[mysql\\_next\\_result](#)

[mysql\\_num\\_fields](#)

[mysql\\_num\\_rows](#)

-O- 

[mysql\\_options](#)

-P- 

[mysql\\_ping](#)

-Q- 

[mysql\\_query](#)

-R- 

[mysql\\_real\\_connect](#)

[mysql\\_real\\_escape\\_string](#)

[mysql\\_real\\_query](#)

[mysql\\_reload](#)

[MYSQL\\_RES](#)

[mysql\\_rollback](#)

MYSQL\_ROW

mysql\_row\_seek

MYSQL\_ROWS

mysql\_row\_tell

**-S-** 

mysql\_select\_db

mysql\_set\_server\_option

mysql\_shutdown

mysql\_sqlstate

mysql\_ssl\_set

mysql\_stat

mysql\_store\_result

**-T-** 

mysql\_thread\_id

**-U-** 

mysql\_use\_result

**-W-** 

mysql\_warning\_count

## Tipo MYSQL

```

typedef struct st_mysql {
    NET                net;                /* Communication parameters */
    gptr               connector_fd;       /* ConnectorFd for SSL */
    char               *host,*user,*passwd,*unix_socket,*server_version,
*host_info,*info;
    char               *db;
    struct charset_info_st *charset;
    MYSQL_FIELD       *fields;
    MEM_ROOT           field_alloc;
    my_ulonglong       affected_rows;
    my_ulonglong       insert_id;         /* id if insert on table with
NEXTNR */
    my_ulonglong       extra_info;        /* Used by mysqlshow */
    unsigned long      thread_id;        /* Id for connection in server */
    unsigned long      packet_length;
    unsigned int       port;
    unsigned long      client_flag,server_capabilities;
    unsigned int       protocol_version;
    unsigned int       field_count;
    unsigned int       server_status;
    unsigned int       server_language;
    unsigned int       warning_count;
    struct st_mysql_options options;
    enum mysql_status  status;
    my_bool            free_me;          /* If free in mysql_close */
    my_bool            reconnect;       /* set to 1 if automatic reconnect
*/

    /* session-wide random string */
    char               scramble[SCRAMBLE_LENGTH+1];

    /*
    Set if this is the original connection, not a master or a slave we have
    added though mysql_rpl_probe() or mysql_set_master()/mysql_add_slave()
    */
    my_bool            rpl_pivot;
    /*
    Pointers to the master, and the next slave connections, points to
    itself if lone connection.
    */
    struct st_mysql*   master, *next_slave;

    struct st_mysql*   last_used_slave; /* needed for round-robin slave pick */

```

```
/* needed for send/read/store/use result to work correctly with
replication */
struct st_mysql* last_used_con;

LIST *stmts; /* list of all statements */
const struct st_mysql_methods *methods;
void *thd;
/*
Points to boolean flag in MYSQL_RES or MYSQL_STMT. We set this flag
from mysql_stmt_close if close had to cancel result set of this object.
*/
my_bool *unbuffered_fetch_owner;
} MYSQL;
```

# Función `mysql_affected_rows()`

```
my_ulonglong mysql_affected_rows(MYSQL *mysql)
```

Devuelve el número de filas afectadas por la última sentencia [UPDATE](#), las borradas por la última sentencia [DELETE](#) o insertadas por la última sentencia [INSERT](#). Debe ser llamada inmediatamente después de la llamada a [mysql\\_query\(\)](#) para las sentencias [UPDATE](#), [DELETE](#) o [INSERT](#). Para sentencias [SELECT](#), `mysql_affected_rows()` funciona igual que [mysql\\_num\\_rows\(\)](#).

## Valores de retorno

Un entero mayor que cero indica el número de filas afectadas o recuperadas. Cero indica que ninguna fila fue actualizada para una sentencia [UPDATE](#), que no hay filas coincidentes con la cláusula *WHERE* de la consulta o que no se ha ejecutado ninguna consulta. -1 indica que la consulta ha retornado un error o que, para una consulta [SELECT](#), `mysql_affected_rows()` fue llamada antes de una llamada a la función [mysql\\_store\\_result\(\)](#). Como `mysql_affected_rows()` devuelve un valor sin signo, se puede verificar el valor -1 comparando el valor de retorno con  $(\text{my\_ulonglong})-1$  (o con  $(\text{my\_ulonglong})\sim 0$ , que es equivalente).

## Errores

No tiene.

## Ejemplo

```
mysql_query(&mysql, "UPDATE products SET cost=cost*1.25 WHERE group=10");  
printf("%ld products updated", (long) mysql_affected_rows(&mysql));
```

Si se especifica la opción **CLIENT\_FOUND\_ROWS** cuando se conecta a `mysqld`, `mysql_affected_rows()` devolverá el número de filas coincidentes con la cláusula *WHERE* para la sentencia [UPDATE](#).

Hay que tener en cuenta que si se usa un comando [REPLACE](#), `mysql_affected_rows()` devuelve 2 si la nueva fila reemplaza a una antigua. Esto es porque en ese caso la fila es insertada después de que la duplicada sea borrada.

Si se usa [INSERT ... ON DUPLICATE KEY UPDATE](#) para insertar una fila, **mysql\_affected\_rows()** devuelve 1 si la fila es insertada como una nueva y 2 si se actualiza una fila existente.

## Función `mysql_autocommit()`

```
my_bool mysql_autocommit(MYSQL *mysql, my_bool mode)
```

Activa el modo *autocommit* si *mode* es 1, o lo desactiva si *mode* es 0.

Esta función se añadió en MySQL 4.1.0.

### Valores de retorno

Cero si tiene éxito. Distinto de cero si se produce un error.

### Errores

Ninguno.

# Función `mysql_change_user()`

```
my_bool mysql_change_user(MYSQL *mysql, const char *user, const char
*password, const char *db)
```

Cambia el usuario y hace que la base de datos especificada por *db* sea la base de datos por defecto (la actual) en la conexión especificada por *mysql*. En consultas sucesivas, esta base de datos será la base de datos por defecto para referencias a tablas que no incluyan un especificador de base de datos explícito.

Esta función se añadió en MySQL 3.23.3.

**`mysql_change_user()`** falla si el usuario conectado no puede ser autenticado o si no tiene permiso para usar las base de datos. En ese caso, el usuario y la base de datos no se cambian.

El parámetro *db* debe ser *NULL* si no se quiere tener una base de datos por defecto.

A partir de MySQL 4.0.6 este comando producirá siempre un **ROLLBACK** en cualquier transacción activa, cierra todas las tablas temporales, desbloquea todas las tablas bloqueadas y resetea el estado como si se hubiese hecho una nueva conexión. Esto ocurrirá aunque el usuario no se cambie.

## Valores de retorno

Cero si tiene éxito. Distinto de cero si ocurre algún error.

## Errores

Los mismos que se obtienen de la función [mysql\\_real\\_connect\(\)](#).

**CR\_COMMANDS\_OUT\_OF\_SYNC**: Los comandos fueron ejecutados en un orden incorrecto.

**CR\_SERVER\_GONE\_ERROR**: El servidor MySQL no está presente.

**CR\_SERVER\_LOST**: La conexión con el servidor se ha perdido durante la consulta.

**CR\_UNKNOWN\_ERROR**: Se ha producido un error desconocido.

**ER\_UNKNOWN\_COM\_ERROR**: El servidor MySQL no implementa este comando (probablemente es

un servidor antiguo).

ER\_ACCESS\_DENIED\_ERROR: El usuario o la contraseña son incorrectos.

ER\_BAD\_DB\_ERROR: La base de datos no existe.

ER\_DBACCESS\_DENIED\_ERROR: El usuario no tiene derechos de acceso a la base de datos.

ER\_WRONG\_DB\_NAME: El nombre de la base de datos es demasiado largo.

## Ejemplo

```
if (mysql_change_user(&mysql, "user", "password", "new_database"))
{
    fprintf(stderr, "Imposible cambiar de usuario. Error: %s\n",
            mysql_error(&mysql));
}
```

## Función `mysql_character_set_name()`

```
const char *mysql_character_set_name(MYSQL *mysql)
```

Devuelve el conjunto de caracteres por defecto para la conexión actual.

### Valores de retorno

El conjunto de caracteres por defecto.

### Errores

No tiene.

# Función `mysql_close()`

```
void mysql_close(MYSQL *mysql);
```

Cierra una conexión previamente abierta. **mysql\_close** también libera de memoria el manipulador apuntado por *mysql* si ese manipulador fue creado automáticamente por [mysql\\_init](#) o [mysql\\_connect](#).

## Parámetros

- **mysql**: dirección de una estructura [MYSQL](#) existente.

## Valor de retorno

No tiene

## Errores

No se pueden producir.

# Función `mysql_commit()`

```
my_bool mysql_commit(MYSQL *mysql)
```

Acomete la transacción actual.

Esta función se añadió en MySQL 4.1.0.

## Valores de retorno

Cero si tiene éxito, un valor distinto de cero si se produce un error.

## Errores

Ninguno.

# Función `mysql_connect()`

```
MYSQL *mysql_connect(MYSQL *mysql, const char *host, const char *usuario,
const char *password);
```

Esta función está fuera de uso, es preferible usar [mysql\\_real\\_connect](#).

**mysql\_connect** intenta establecer una conexión con un motor de bases de datos MySQL ejecutándose en *host*. **mysql\_connect** debe completarse con éxito antes de que se pueda ejecutar cualquier otra función del API, con excepción de la función [mysql\\_get\\_client\\_info](#).

El significado de los parámetros es el mismo que para los correspondientes en la función [mysql\\_real\\_connect](#) con la diferencia de que el parámetro *mysql* puede ser NULL. En ese caso el API reserva memoria de forma automática para la estructura [connection](#) y la libera cuando se llama a [mysql\\_close](#). La desventaja de esta solución es que no es posible obtener un mensaje de error si la conexión falla. (Para obtener información de errores con las funciones [mysql\\_errno](#) o [mysql\\_error](#), se debe proporcionar un puntero [MYSQL](#) válido.)

## Valor de retorno

El mismo que para [mysql\\_real\\_connect](#).

## Errores

Los mismos que para [mysql\\_real\\_connect](#).

# Función `mysql_create_db()`

```
int mysql_create_db(MYSQL *mysql, const char *database);
```

Crea la base de datos con el nombre del parámetro *database*.

Esta función está desaconsejada. Es preferible usar [mysql\\_query](#) para ejecutar una instrucción SQL [CREATE DATABASE](#) en su lugar.

## Parámetros

- **mysql**: El primer parámetro debe ser la dirección de una estructura [MYSQL](#) existente.
- **database**: nombre de la base de datos a crear.

## Valor de retorno

El valor de retorno es cero si la base de datos fue creada correctamente. Un valor distinto de cero indica que ha ocurrido un error.

## Errores

CR\_COMMANDS\_OUT\_OF\_SYNC: los comandos fueron ejecutados en un orden inapropiado.

CR\_SERVER\_GONE\_ERROR: el servidor MySQL no está presente.

CR\_SERVER\_LOST: la conexión con el servidor se perdió durante la consulta.

CR\_UNKNOWN\_ERROR: ha ocurrido un error desconocido.

## Ejemplo

```
if(mysql_create_db(&mysql, "my_database"))
{
    fprintf(stderr, "Imposible crear la nueva base de datos. Error: %s\n",
            mysql_error(&mysql));
}
```



## Tipo MYSQL\_DATA

```
typedef struct st_mysql_data {
    my_ulonglong rows;
    unsigned int fields;
    MYSQL_ROWS *data;
    MEM_ROOT alloc;
#ifdef !defined(CHECK_EMBEDDED_DIFFERENCES) || defined(EMBEDDED_LIBRARY)
    MYSQL_ROWS **prev_ptr;
#endif
} MYSQL_DATA;
```

## Función `mysql_data_seek()`

```
void mysql_data_seek(MYSQL_RES *result, my_ulonglong offset)
```

Busca una fila arbitraria en un conjunto de resultados de una consulta. El valor de *offset* o desplazamiento es un número de fila y debe estar en el rango de 0 a [mysql\\_num\\_rows\(result\)-1](#).

Esta función requiere que la estructura del conjunto de resultados contenga el resultado entero de la consulta, de modo que **mysql\_data\_seek()** puede ser usado sólo junto con [mysql\\_store\\_result\(\)](#), no con [mysql\\_use\\_result\(\)](#).

### Valores de retorno

Ninguno.

### Errores

Ninguno

## Función `mysql_debug()`

```
void mysql_debug(const char *debug)
```

Realiza un *DEBUG\_PUSH* con la cadena dada. `mysql_debug()` usa la librería de depuración **Fred Fish debug**. Para usar esta función, se debe compilar la librería del cliente para que soporte depuración.

### Valores de retorno

Ninguno.

### Errores

Ninguno.

### Ejemplo

La llamada mostrada aquí hace que la librería de cliente genere una traza en el fichero `'/tmp/client.trace'` de la máquina actual:

```
mysql_debug("d:t:0,/tmp/client.trace");
```

## Función `mysql_drop_db()`

```
int mysql_drop_db(MYSQL *mysql, const char *db)
```

Elimina la base de datos con el nombre *db*.

Esta función está desaconsejada. Es preferible usar [mysql\\_query\(\)](#) para lanzar una sentencia SQL [DROP DATABASE](#) en su lugar.

### Valores de retorno

Cero si la base de datos fue eliminada. Distinto de cero si se produjo un error.

### Errores

CR\_COMMANDS\_OUT\_OF\_SYNC: Los comandos fueron ejecutados en un orden incorrecto.

CR\_SERVER\_GONE\_ERROR: El servidor MySQL no está presente.

CR\_SERVER\_LOST: La conexión con el servidor se ha perdido durante la consulta.

CR\_UNKNOWN\_ERROR: Se ha producido un error desconocido.

### Ejemplo

```
if(mysql_drop_db(&mysql, "my_database"))
    fprintf(stderr, "Error al eliminar la base de datos: Error: %s\n",
            mysql_error(&mysql));
```

# Función `mysql_dump_debug_info()`

```
int mysql_dump_debug_info(MYSQL *mysql)
```

Indica al servidor que escriba alguna información de depuración en el diario. Para que funcione, el usuario conectado debe tener el privilegio *SUPER*.

## Valores de retorno

Cero si el comando se ha ejecutado correctamente. Distinto de cero si se produjo un error.

## Errores

**CR\_COMMANDS\_OUT\_OF\_SYNC:** Los comandos fueron ejecutados en un orden incorrecto.

**CR\_SERVER\_GONE\_ERROR:** El servidor MySQL no está presente.

**CR\_SERVER\_LOST:** La conexión con el servidor se ha perdido durante la consulta.

**CR\_UNKNOWN\_ERROR:** Se ha producido un error desconocido.

# Función `mysql_eof()`

```
my_bool mysql_eof(MYSQL_RES *result)
```

Esta función está desaconsejada. [mysql\\_errno\(\)](#) o [mysql\\_error\(\)](#) pueden usarse en su lugar.

**mysql\_eof()** determina si la última fila de un conjunto de resultados ha sido leída.

Si se obtiene un conjunto de resultados a partir de una llamada exitosa a [mysql\\_store\\_result\(\)](#), el cliente recibe el conjunto completo en una operación. En ese caso, un valor *NULL* de una llamada a [mysql\\_fetch\\_row\(\)](#) siempre significa que el final del conjunto de resultados ha sido alcanzado y es innecesario llamar a **mysql\_eof()**. Cuando se usa con [mysql\\_store\\_result\(\)](#), **mysql\_eof()** siempre devuelve verdadero.

Por otra parte, si se usa [mysql\\_use\\_result\(\)](#) para iniciar una recuperación de un conjunto de resultados, las filas del conjunto se obtienen del servidor una a una como si se hubiese llamado a [mysql\\_fetch\\_row\(\)](#) repetidamente. Debido a que se puede producir un error en la conexión durante este proceso, un valor de retorno *NULL* desde [mysql\\_fetch\\_row\(\)](#) no significa necesariamente que en final del conjunto de resultados ha sido alcanzado con normalidad. En ese caso, se puede usar **mysql\_eof()** para determinar qué ha sucedido. **mysql\_eof()** devuelve un valor distinto de cero si se ha alcanzado el final del conjunto de resultados y cero si ha ocurrido un error.

Historicamente, **mysql\_eof()** precede a las funciones de error estándar de MySQL [mysql\\_errno\(\)](#) y [mysql\\_error\(\)](#). Ya que estas funciones de error proporcionan la misma información, su uso es preferible sobre **mysql\_eof()**, que actualmente está desaconsejada. (De hecho, esas funciones proporcionan más información, ya que **mysql\_eof()** devuelve sólo un valor booleano mientras que las funciones de error indican un motivo para el error cuando uno ocurre.)

## Valores de retorno

Cero si no ha ocurrido un error. Distinto de cero si se ha alcanzado el final del conjunto de resultados.

## Errores

Ninguno

## Ejemplo

El siguiente ejemplo muestra cómo se debe usar `mysql_eof()`:

```
mysql_query(&mysql, "SELECT * FROM some_table");
result = mysql_use_result(&mysql);
while((row = mysql_fetch_row(result)))
{
    // hacer algo con los datos
}
if(!mysql_eof(result)) // mysql_fetch_row() fallo debido a un error
{
    fprintf(stderr, "Error: %s\n", mysql_error(&mysql));
}
```

Sin embargo, se puede conseguir el mismo efecto con las funciones de error estándar de MySQL:

```
mysql_query(&mysql, "SELECT * FROM some_table");
result = mysql_use_result(&mysql);
while((row = mysql_fetch_row(result)))
{
    // hacer algo con los datos
}
if(mysql_errno(&mysql)) // mysql_fetch_row() fallo debido a un error
{
    fprintf(stderr, "Error: %s\n", mysql_error(&mysql));
}
```

# Función `mysql_errno()`

```
unsigned int mysql_errno(MYSQL *mysql);
```

Para la conexión especificada por *mysql*, devuelve el código de error para la función del API invocada más recientemente, tanto si tuvo éxito como si no. Un valor de retorno cero significa que no ocurrió un error.

Los números de mensajes de error del cliente se listan en el fichero de cabecera 'errmsg.h'. Los del servidor en 'mysql\_error.h'. En el fichero de distribución de MySQL se puede encontrar una lista completa de mensajes de error y números de error en el fichero 'Docs/mysql\_error.txt'.

Hay algunas funciones, como [mysql\\_fetch\\_row](#) que no activan el número de error si tienen éxito.

Una regla para esto es que todas las funciones que tienen que preguntar al servidor por información resetean el número de error si tienen éxito.

## Parámetros

- **mysql**: El primer parámetro debe ser la dirección de una estructura [MYSQL](#) existente.

## Valor de retorno

Un valor de código de error para la última llamada a una función `mysql_xxx`, si ha fallado. Cero significa que no ha ocurrido un error.

## Función `mysql_error()`

```
const char *mysql_error(MYSQL *mysql)
```

Para la conexión especificada por *mysql*, **mysql\_error()** devuelve una cadena terminada en cero consistente en el mensaje de error para la invocación más reciente de una función del API que haya fallado. Si ninguna función ha fallado, el valor de retorno de **mysql\_error()** puede ser el error previo o una cadena vacía para indicar que no hay error.

Una regla para esto es que todas las funciones que tienen que preguntar al servidor por información resetean el número de error si tienen éxito.

Para funciones que anulen el valor de [mysql\\_errno\(\)](#), las dos comprobaciones siguientes son equivalentes:

```
if(mysql_errno(&mysql))
{
    // se ha producido un error
}

if(mysql_error(&mysql)[0] != '\0')
{
    // se ha producido un error
}
```

El lenguaje de los mensajes de error del cliente puede cambiarse recompilando la librería del cliente MySQL. Actualmente no es posible elegir mensajes de error en varios lenguajes distintos.

### Valores de retorno

Una cadena terminada en cero que describe el error. Una cadena vacía si no se ha producido un error.

### Errores

Ninguno.

## ***Función `mysql_escape_string()`***

Se debe usar la función [`mysql\_real\_escape\_string\(\)`](#) en su lugar.

Esta función es idéntica a [`mysql\_real\_escape\_string\(\)`](#) excepto que [`mysql\_real\_escape\_string\(\)`](#) toma un manipulador de conexión como primer argumento y escapa la cadena de acuerdo con el conjunto de caracteres actual. **`mysql_escape_string()`** no requiere un argumento de conexión y no tiene en cuenta la asignación actual del conjunto de caracteres.

# Función `mysql_fetch_field()`

```
MYSQL_FIELD *mysql_fetch_field(MYSQL_RES *result)
```

Devuelve la definición de una columna de un conjunto de resultados como una estructura `MYSQL_FIELD`. Hay que llamar a esta función repetidamente para recuperar la información sobre todas las columnas del conjunto de resultados. `mysql_fetch_field()` devuelve `NULL` cuando no quedan más campos.

`mysql_fetch_field()` se resetea para devolver la información sobre la primera columna cada vez que se ejecute una nueva consulta `SELECT`. El campo que se devolverá por `mysql_fetch_field()` también se ve afectado por llamadas a `mysql_field_seek()`.

Si se ha llamado a `mysql_query()` para realizar un `SELECT` en una tabla pero no se ha llamado a `mysql_store_result()`, MySQL devuelve el tamaño por defecto de bloque (8KB) si se llama a `mysql_fetch_field()` para preguntar por la longitud de una columna `BLOB`. (El tamaño de 8KB se elige porque MySQL no conoce el tamaño máximo para el `BLOB`. Esto se hará configurable en el futuro.) Una vez que se ha recuperado el conjunto de resultados, `field->max_length` contiene la longitud del valor más largo para esa columna en la consulta específica.

## Valores de retorno

La estructura `MYSQL_FIELD` para la columna actual. `NULL` si no quedan columnas por recuperar.

## Errores

Ninguno.

## Ejemplo

```
MYSQL_FIELD *field;

while((field = mysql_fetch_field(result)))
{
    printf("Nombre de campo %s\n", field->name);
}
```

# Función `mysql_fetch_fields()`

```
MYSQL_FIELD *mysql_fetch_fields(MYSQL_RES *result)
```

Devuelve un array con todas las estructuras [MYSQL\\_FIELD](#) para un conjunto de resultados. Cada estructura proporciona una definición de campo para una columna del conjunto de resultados.

## Valores de retorno

Un array de estructuras [MYSQL\\_FIELD](#) para todas las columnas del conjunto de resultados.

## Errores

Ninguno.

## Ejemplo

```
unsigned int num_fields;
unsigned int i;
MYSQL_FIELD *fields;

num_fields = mysql_num_fields(result);
fields = mysql_fetch_fields(result);
for(i = 0; i < num_fields; i++)
{
    printf("Columna %u es %s\n", i, fields[i].name);
}
```

# Función `mysql_fetch_field_direct()`

```
MYSQL_FIELD *mysql_fetch_field_direct(MYSQL_RES *result, unsigned int fieldnr)
```

Dado un número de campo *fieldnr* para una columna dentro de un conjunto de resultados, devuelve la definición de esa columna como una estructura [MYSQL\\_FIELD](#). Se debe usar esta función para decuperar la definición de una columna arbitraria. El valor de *fieldnr* debe estar en el rango de 0 a `mysql_num_fields(result)-1`.

## Valores de retorno

La estructura [MYSQL\\_FIELD](#) para la columna especificada.

## Errores

Ninguno.

## Ejemplo

```
unsigned int num_fields;
unsigned int i;
MYSQL_FIELD *field;

num_fields = mysql_num_fields(result);
for(i = 0; i < num_fields; i++)
{
    field = mysql_fetch_field_direct(result, i);
    printf("Columna %u es %s\n", i, field->name);
}
```

# Función `mysql_fetch_lengths()`

```
unsigned long *mysql_fetch_lengths(MYSQL_RES *result)
```

Devuelve las longitudes de las columnas de la fila actual de un conjunto de resultados. Si se planea copiar valores de campos, esta información de longitud es muy útil para la optimización, ya que se puede evitar la llamada a `strlen()`. Además, si el conjunto de resultados contiene datos binarios, se debe usar esta función para determinar el tamaño de los datos, porque `strlen()` devolverá resultados incorrectos para cualquier campo que contenga caracteres nulos.

La longitud para cadenas vacías y para columnas que contengan valores *NULL* es cero. Para distinguir entre ambos casos, ver la descripción de la función [mysql\\_fetch\\_row\(\)](#).

## Valores de retorno

Un array de enteros *long* sin signo que representas los tamaños de cada columna (sin incluir el carácter nulo terminador). *NULL* si se produce un error.

## Errores

`mysql_fetch_lengths()` sólo es válido para la fila actual del conjunto de resultados. Devuelve *NULL* si es llamada antes de llamar a [mysql\\_fetch\\_row\(\)](#) o después de recuperar todas las filas del resultado.

## Ejemplo

```
MYSQL_ROW row;
unsigned long *lengths;
unsigned int num_fields;
unsigned int i;

row = mysql_fetch_row(result);
if (row)
{
    num_fields = mysql_num_fields(result);
    lengths = mysql_fetch_lengths(result);
    for(i = 0; i < num_fields; i++)
    {
        printf("Columna %u tiene %lu bytes de longitud.\n", i, lengths
```

```
[ i ] ) ;  
    }  
}
```

# Función `mysql_fetch_row()`

```
MYSQL_ROW mysql_fetch_row(MYSQL_RES *result);
```

Recupera la siguiente fila de un conjunto de resultados. Cuando se usa después de [mysql\\_store\\_result](#), **mysql\_fetch\_row** devuelve *NULL* si no quedan más filas por recuperar. Cuando se usa después de [mysql\\_use\\_result](#), [mysql\\_fetch\\_row](#) devuelve *NULL* si no quedan filas por recuperar o si se ha producido un error.

El número de valores en la fila viene dado por [mysql\\_num\\_fields\(result\)](#). Si la fila contiene el valor devuelto por una llamada a **mysql\_fetch\_row**, los punteros a los valores serán accesibles como `row[0]` a `row[mysql_num_fields(result)-1]`. Los valores *NULL* en la fila se indican mediante punteros nulos.

Las longitudes de los valores de los campos en la fila se pueden obtener mediante una llamada a la función [mysql\\_fetch\\_lengths](#). Los campos vacíos y los que contengan *NULL* tendrán longitud 0; se pueden distinguir comprobando el puntero para el valor del campo. Si el puntero es *NULL*, el campo es *NULL*; en otro caso, el campo estará vacío.

## Valores de retorno

Una estructura [MYSQL\\_ROW](#) para la siguiente fila. *NULL* si no hay más filas para recuperar o si se ha producido un error.

## Errores

Los errores no se resetean entre sucesivas llamadas a **mysql\_fetch\_row**.

CR\_SERVER\_LOST: la conexión con el servidor se perdió durante la consulta.

CR\_UNKNOWN\_ERROR: ha ocurrido un error desconocido.

## Ejemplo

```
MYSQL_ROW row;
unsigned int num_fields;
unsigned int i;
```

```
num_fields = mysql_num_fields(result);
while ((row = mysql_fetch_row(result))) {
    unsigned long *lengths;
    lengths = mysql_fetch_lengths(result);
    for(i = 0; i < num_fields; i++) {
        printf("[%.*s] ", (int) lengths[i], row[i] ? row[i] : "NULL");
    }
    printf("\n");
}
```

## Tipo MYSQL\_FIELD

```
typedef struct st_mysql_field {
    char *name;                /* Name of column */
    char *org_name;            /* Original column name, if an alias */
    char *table;               /* Table of column if column was a field */
    char *org_table;           /* Org table name, if table was an alias */
    char *db;                  /* Database for table */
    char *catalog;             /* Catalog for table */
    char *def;                 /* Default value (set by mysql_list_fields)
*/
    unsigned long length;      /* Width of column (create length) */
    unsigned long max_length;  /* Max width for selected set */
    unsigned int name_length;
    unsigned int org_name_length;
    unsigned int table_length;
    unsigned int org_table_length;
    unsigned int db_length;
    unsigned int catalog_length;
    unsigned int def_length;
    unsigned int flags;        /* Div flags */
    unsigned int decimals;     /* Number of decimals in field */
    unsigned int charsetnr;    /* Character set */
    enum enum_field_types type; /* Type of field. See mysql_com.h for types
*/
} MYSQL_FIELD;
```

# Función `mysql_field_count()`

```
unsigned int mysql_field_count(MYSQL *mysql)
```

Si se está usando una versión de MySQL previa a la 3.22.24, se debe usar en su lugar `unsigned int mysql\_num\_fields(MYSQL *mysql)`.

Devuelve el número de columnas para la consulta más reciente de la conexión *mysql*.

El uso habitual de esta función es cuando `mysql\_store\_result\(\)` devuelve *NULL* (y entonces no se dispone de un puntero al conjunto de resultados). En ese caso, se puede llamar a `mysql\_field\_count\(\)` para determinar si `mysql\_store\_result\(\)` debería haber producido un resultado no vacío. Esto permite al programa cliente tomar las acciones adecuadas sin saber si la consulta fue una sentencia `SELECT` (o parecida). El ejemplo mostrado aquí ilustra como puede hacerse esto.

Ver `mysql\_store\_result\(\)` para ver por qué a veces esta función devuelve *NULL* después de que `mysql\_query\(\)` retorne con éxito.

## Valores de retorno

Un entero sin signo que representa el número de columnas en el conjunto de resultados.

## Errores

Ninguno

## Ejemplo

```
MYSQL_RES *result;
unsigned int num_fields;
unsigned int num_rows;

if (mysql\_query(&mysql, query_string))
{
    // error
}
else // consulta exitosa, procesar cualquier dato retornado
```

```

{
    result = mysql_store_result(&mysql);
    if (result) // there are rows
    {
        num_fields = mysql_num_fields(result);
        // recuperar filas, y después llamar a mysql_free_result(result)
    }
    else // mysql_store_result() no ha devuelto nada; ¿debería haberlo
hecho?
    {
        if(mysql_field_count(&mysql) == 0)
        {
            // la consulta no devuelve datos
            // (no fue un SELECT)
            num_rows = mysql_affected_rows(&mysql);
        }
        else // mysql_store_result() ha devuelto datos
        {
            fprintf(stderr, "Error: %s\n", mysql_error(&mysql));
        }
    }
}

```

Una alternativa consiste en reemplazar la llamada a mysql\_field\_count(&mysql) con mysql\_errno(&mysql). En ese caso, se estará comprobando de forma directa un error de mysql\_store\_result() en lugar de deducir del valor de retorno de mysql\_field\_count() si la sentencia fue un SELECT.

## Función `mysql_field_seek()`

```
MYSQL_FIELD_OFFSET mysql_field_seek(MYSQL_RES *result, MYSQL_FIELD_OFFSET offset)
```

Asigna al cursor de campo el desplazamiento *offset*. La siguiente llamada a [mysql\\_fetch\\_field\(\)](#) recuperará la definición de campo de la columna asociada con ese *offset*.

Para colocar el cursor al principio de la fila, hay que indicar un desplazamiento de cero.

### Valores de retorno

El valor previo del cursor de campo.

### Errores

Ninguno.

# Función `mysql_field_tell()`

```
MYSQL_FIELD_OFFSET mysql_field_tell(MYSQL_RES *result)
```

Devuelve la posición del cursor de campo usado por la última llamada a [mysql\\_fetch\\_field\(\)](#). Este valor puede ser usado como argumento para la función [mysql\\_field\\_seek\(\)](#).

## Valores de retorno

El valor actual de desplazamiento del cursor de campo.

## Errores

Ninguno.

# Función `mysql_free_result()`

```
void mysql_free_result(MYSQL_RES *result);
```

Libera la memoria reservada para un conjunto de resultados *result* por una función [mysql\\_store\\_result](#), [mysql\\_use\\_result](#), [mysql\\_list\\_dbs](#), etc. Cuando se haya terminado de trabajar con un conjunto de resultados, se debe liberar la memoria que usa mediante una llamada a **mysql\_free\_result**.

No se debe intentar acceder a un conjunto de resultados después de haberlo liberado.

## Valores de retorno

No tiene

## Errores

No se pueden producir.

## Función `mysql_get_client_info()`

```
char *mysql_get_client_info(void)
```

Devuelve una cadena que representa la versión de la librería de cliente.

### Valores de retorno

Una cadena de caracteres que representa la versión de la librería de cliente MySQL.

### Errores

Ninguno.

# Función `mysql_get_client_version()`

```
unsigned long mysql_get_client_version(void)
```

Devuelve un entero que representa la versión de la librería de cliente. El valor tiene el formato `XYYZZ`, donde `X` es la versión, `YY` es el nivel de *release* y `ZZ` es el número de versión dentro del nivel de *release*. Por ejemplo, un valor `40102` representa una versión de librería de cliente `4.1.2`.

Esta función fue añadida en MySQL 4.0.16.

## Valores de retorno

Un entero que representa la versión de librería de cliente MySQL.

## Errores

Ninguno.

## Función `mysql_get_host_info()`

```
char *mysql_get_host_info(MYSQL *mysql)
```

Devuelve una cadena que describe el tipo de conexión actual, incluyendo el nombre del servidor.

### Valores de retorno

Una cadena de caracteres que representa el nombre del servidor y el tipo de conexión.

### Errores

Ninguno.

## Función `mysql_get_proto_info()`

```
unsigned int mysql_get_proto_info(MYSQL *mysql)
```

Devuelve la versión de protocolo para la conexión actual.

### Valores de retorno

Un valor entero sin signo que represente la versión de protocolo usada por la conexión actual.

### Errores

Ninguno

## Función `mysql_get_server_info()`

```
char *mysql_get_server_info(MYSQL *mysql)
```

Devuelve una cadena que representa el número de versión del servidor.

### Valores de retorno

Una cadena de caracteres que representa el número de versión del servidor.

### Errores

Ninguno.

## Función `mysql_get_server_version()`

```
unsigned long mysql_get_server_version(MYSQL *mysql)
```

Devuelve el número de versión del servidor como un entero.

Esta función se añadió en MySQL 4.1.0.

### Valores de retorno

Un número que representa la versión del servidor MySQL en este formato:

```
major_version*10000 + minor_version *100 + sub_version
```

Por ejemplo, 4.1.2 se devuelve como 40102.

Esta función es práctica en programas cliente para determinar rápidamente si existe alguna capacidad específica de determinada versión.

### Errores

Ninguno.

## Función `mysql_hex_string()`

```
unsigned long mysql_hex_string(char *to, const char *from, unsigned long
length)
```

Esta función se usa para crear una cadena SQL legal que se puede usar en una sentencia SQL.

La cadena *from* se codifica a formato hexadecimal, con cada carácter codificado como dos dígitos hexadecimales. El resultado se coloca en la cadena *to* y se añade el carácter nulo terminador.

La cadena apuntada por *to* debe tener *length* bytes de longitud. Se debe conseguir memoria para el buffer *to* para que tenga al menos  $length*2+1$  bytes de longitud. Cuando `mysql_hex_string()` regresa, el contenido de *to* será una cadena terminada con nulo. El valor de retorno será la longitud de la cadena codificada, sin incluir el carácter nulo terminador.

El valor de retorno puede ser colocado en una sentencia SQL usando tanto el formato `0xvalor` o `X'valor'`. Sin embargo, el valor devuelto no incluye el `0x` o `X'...'`. El usuario debe añadir cualquiera de ambos, tal como desee.

`mysql_hex_string()` fue añadido en MySQL 4.0.23 y 4.1.8.

### Ejemplo

```
char query[1000],*end;

end = strmov(query,"INSERT INTO test_table values(");
end = strmov(end,"0x");
end += mysql_hex_string(end,"Qué es esto",11);
end = strmov(end,",0x");
end += mysql_hex_string(end,"datos binarios: \0\r\n",19);
*end++ = ')';

if (mysql_real_query(&mysql,query,(unsigned int) (end - query)))
{
    fprintf(stderr, "Error al insertar fila, Error: %s\n",
            mysql_error(&mysql));
}
```

La función `strmov()` usada en este ejemplo se incluye en la librería `mysqlclient` y funciona como `strcpy`

() pero devolviendo un puntero al terminador nulo del primer parámetro.

## Valores de retorno

La longitud del valor colocado en *to*, sin incluir el carácter nulo terminador.

## Errores

Ninguno.

# mysql\_info()

```
char *mysql_info(MYSQL *mysql)
```

Recupera una cadena que proporciona información sobre la consulta ejecutada más recientemente, pero sólo para las sentencias listadas aquí. Para otras sentencias, **mysql\_info()** devuelve *NULL*. El formato de la cadena varía dependiendo del tipo de consulta, tal como se describe. Los números son sólo ilustrativos; la cadena contendrá valores apropiados para cada consulta.

## INSERT INTO ... SELECT ...

Formato de cadena: `Records: 100 Duplicates: 0 Warnings: 0`

## INSERT INTO ... VALUES (...),(...),(...)...

Formato de cadena: `Records: 3 Duplicates: 0 Warnings: 0`

## LOAD DATA INFILE ...

Formato de cadena: `Records: 1 Deleted: 0 Skipped: 0 Warnings: 0`

## ALTER TABLE

Formato de cadena: `Records: 3 Duplicates: 0 Warnings: 0`

## UPDATE

Formato de cadena: `Rows matched: 40 Changed: 40 Warnings: 0`

Hay que tener en cuenta que **mysql\_info()** devuelve un valor no nulo para INSERT ... VALUES sólo para el formato de la sentencia de múltiples filas (es decir, sólo si se especifica una lista de múltiples valores).

## Valores de retorno

Una cadena de caracteres que representa información adicional sobre la consulta ejecutada más recientemente. *NULL* si no hay información disponible para la consulta.

## Errores

Ninguno.

# Función `mysql_init()`

```
MYSQL *mysql_init(MYSQL *mysql);
```

Crea o inicializa un objeto [MYSQL](#), que posteriormente puede ser usado por la función [mysql\\_real\\_connect](#). Si el parámetro es NULL, la función crea, inicializa y devuelve un objeto nuevo. En otro caso, el objeto es inicializado y se devuelve su dirección. Si la función crea un objeto nuevo, será liberado cuando se invoque a la función [mysql\\_close](#) para cerrar la conexión.

Para evitar pérdidas de memoria, usar el procedimiento siguiente, que debe ser hecho cada vez que la aplicación se enlace con la librería *libmysqlclient* o *libmysqld*:

- Llamar a [mysql\\_library\\_init\(\)](#) antes de la primera llamada a **mysql\_init()**.
- Llamar a [mysql\\_library\\_end\(\)](#) después de que la aplicación haya cerrado cualquier conexión abierta que haya sido hecha usando el API C de MySQL.
- Si se desea, la llamada a [mysql\\_library\\_init\(\)](#) puede omitirse, porque **mysql\_init()** la invocará automáticamente si es necesario.

## Valor de retorno

Un manipulador inicializado. NULL si no existe memoria suficiente para crear un objeto nuevo.

## Errores

En caso de memoria insuficiente, se devuelve NULL.

# Función `mysql_insert_id()`

```
my_ulonglong mysql_insert_id(MYSQL *mysql)
```

Devuelve el valor generado por una columna `AUTO_INCREMENT` para la sentencia `INSERT` o `UPDATE` previa. Usar esta función después de realizar una sentencia `INSERT` en una tabla que contenga una columna `AUTO_INCREMENT`.

Más precisamente, `mysql_insert_id()` se actualiza en las siguientes condiciones:

- Sentencias `INSERT` que almacenen un valor en una columna `AUTO_INCREMENT`. Esto será cierto tanto si el valor fue generado automáticamente almacenando los valores especiales `NULL` ó `0` en la columna, o si se ha usado un valor explícito no especial.
- En el caso de una sentencia `INSERT` de múltiples filas, `mysql_insert_id()` devuelve el primer valor `AUTO_INCREMENT` generado automáticamente; si no se a generado ninguno, se devuelve el último valor insertado explícitamente en la columna `AUTO_INCREMENT`.
- Sentencias `INSERT` que generen un valor `AUTO_INCREMENT` mediante la inserción del valor `LAST_INSERT_ID(expr)` en cualquier columna.
- Sentencias `INSERT` que generen un valor `AUTO_INCREMENT` mediante la actualización de cualquier columna con el valor `LAST_INSERT_ID(expr)`.
- El valor de `mysql_insert_id()` no se ve afectado por sentencias como `SELECT`, que devuelvan un conjunto de resultados.
- Si la última sentencia a devuelto un error, el valor de `mysql_insert_id()` está indefinido.

`mysql_insert_id()` devuelve 0 si la sentencia previa no usa un valor `AUTO_INCREMENT`. Si se necesita guardar el valor para usarlo más tarde, asegurarse de llamar a `mysql_insert_id()` inmediatamente después de la sentencia que genera el valor.

El valor de `mysql_insert_id()` se ve afectado sólo por sentencias lanzadas dentro de la conexión de cliente actual. No se ve afectado por sentencias lanzadas por otros clientes.

También hay que tener en cuenta que la función SQL `LAST_INSERT_ID()` siempre contiene el valor `AUTO_INCREMENT` generado más recientemente, y no se resetea entre sentencias porque el valor de esa función es mantenido por el servidor. Otra diferencia es que `LAST_INSERT_ID()` no se actualiza si se asigna a una columna `AUTO_INCREMENT` un valor específico no especial.

El motivo de estas diferencias entre `LAST_INSERT_ID()` y `mysql_insert_id()` es que `LAST_INSERT_ID()` se ha diseñado para que sea fácil de usar en scripts mientras que `mysql_insert_id`

() intenta proporcionar una información algo más exacta de lo que ocurre en una columna *AUTO\_INCREMENT*.

## Valores de retorno

Los descritos anteriormente.

## Errores

Ninguno.

# Función `mysql_kill()`

```
int mysql_kill(MYSQL *mysql, unsigned long pid)
```

Pregunta al servidor para matar un hilo especificado por *pid*.

## Valores de retorno

Cero si tiene éxito. Distinto de cero si ocurre un error.

## Errores

CR\_COMMANDS\_OUT\_OF\_SYNC: los comandos fueron ejecutados en un orden inapropiado.

CR\_SERVER\_GONE\_ERROR: el servidor MySQL no está presente.

CR\_SERVER\_LOST: la conexión con el servidor se perdió durante la última consulta.

CR\_UNKNOWN\_ERROR: se ha producido un error desconocido.

## ***Función `mysql_library_end()`***

```
void mysql_library_end(void)
```

Es un sinónimo de la función [mysql\\_server\\_end\(\)](#). Se añadió en MySQL 4.1.10 y 5.0.3.

## ***Función `mysql_library_init()`***

```
int mysql_library_init(int argc, char **argv, char **groups)
```

Es un sinónimo de la función [mysql\\_server\\_init\(\)](#). Se añadió en MySQL 4.1.10 y 5.0.3.

# Función `mysql_list_dbs()`

```
MYSQL_RES *mysql_list_dbs(MYSQL *mysql, const char *wild)
```

Devuelve un conjunto de resultados consistente en los nombres de las bases de datos en el servidor que coinciden con la expresión regular simple especificada por el parámetro *wild*. *wild* puede contener los caracteres comodín ``%' o `_'`, o puede ser un puntero *NULL* para buscar todas las bases de datos. Llamar a `mysql_list_dbs()` es similar que ejecutar la consulta [SHOW DATABASES \[LIKE wild\]](#).

Se debe liberar el conjunto de resultados mediante [mysql\\_free\\_result\(\)](#).

## Valores de retorno

Un conjunto de resultados [MYSQL\\_RES](#) si tiene éxito. *NULL* si ocurre un error.

## Errores

`CR_COMMANDS_OUT_OF_SYNC`: los comandos fueron ejecutados en un orden inapropiado.

`CR_OUT_OF_MEMORY`: falta memoria.

`CR_SERVER_GONE_ERROR`: el servidor MySQL no está presente.

`CR_SERVER_LOST`: la conexión con el servidor se perdió durante la última consulta.

`CR_UNKNOWN_ERROR`: se ha producido un error desconocido.

# Función `mysql_list_fields()`

```
MYSQL_RES *mysql_list_fields(MYSQL *mysql, const char *table, const char *wild)
```

Devuelve un conjunto de resultados que consiste en los nombres de los campos en la tabla *table*, que coincidan con la expresión regular simple especificada por el parámetro *wild*. *wild* puede contener los caracteres comodín '%' o '\_', o puede ser un puntero *NULL* para obtener todos los campos. Llamar a `mysql_list_fields()` es similar a ejecutar la consulta [SHOW COLUMNS FROM tbl\\_name \[LIKE wild\]](#).

Es mejor usar [SHOW COLUMNS FROM tbl\\_name](#) en lugar de `mysql_list_fields()`.

Se debe liberar el conjunto de resultados usando la función [mysql\\_free\\_result\(\)](#).

## Valores de retorno

Un conjunto de resultados [MYSQL\\_RES](#) si tiene éxito. *NULL* si se produce un error.

## Errores

`CR_COMMANDS_OUT_OF_SYNC`: los comandos fueron ejecutados en un orden inapropiado.

`CR_SERVER_GONE_ERROR`: el servidor MySQL no está presente.

`CR_SERVER_LOST`: la conexión con el servidor se perdió durante la última consulta.

`CR_UNKNOWN_ERROR`: se ha producido un error desconocido.

# Función `mysql_list_processes()`

```
MYSQL_RES *mysql_list_processes(MYSQL *mysql)
```

Devuelve un conjunto de resultados que describen los procesos actuales del servidor. Es el mismo tipo de información que el devuelto por `mysqladmin processlist` o una consulta [SHOW PROCESSLIST](#).

Se debe liberar el conjunto de resultados usando la función [mysql\\_free\\_result\(\)](#).

## Valores de retorno

Un conjunto de resultados [MYSQL\\_RES](#) si tiene éxito. `NULL` si se produce un error.

## Errores

`CR_COMMANDS_OUT_OF_SYNC`: los comandos fueron ejecutados en un orden inapropiado.

`CR_SERVER_GONE_ERROR`: el servidor MySQL no está presente.

`CR_SERVER_LOST`: la conexión con el servidor se perdió durante la última consulta.

`CR_UNKNOWN_ERROR`: se ha producido un error desconocido.

# Función `mysql_list_tables()`

```
MYSQL_RES *mysql_list_tables(MYSQL *mysql, const char *wild)
```

Devuelve un conjunto de resultados que consiste en los nombres de las tablas en la base de datos actual que coinciden con la expresión regular simple especificada por el parámetro *wild*. *wild* puede contener los caracteres comodín '%' o '\_', o puede ser un puntero *NULL* para recuperar todas las tablas. Llamar a `mysql_list_tables()` es similar a ejecutar la consulta [SHOW TABLES \[LIKE wild\]](#).

Se debe liberar el conjunto de resultados usando la función [mysql\\_free\\_result\(\)](#).

## Valores de retorno

Un conjunto de resultados [MYSQL\\_RES](#) si tiene éxito. *NULL* si se produce un error.

## Errores

CR\_COMMANDS\_OUT\_OF\_SYNC: los comandos fueron ejecutados en un orden inapropiado.

CR\_SERVER\_GONE\_ERROR: el servidor MySQL no está presente.

CR\_SERVER\_LOST: la conexión con el servidor se perdió durante la última consulta.

CR\_UNKNOWN\_ERROR: se ha producido un error desconocido.

# Función `mysql_more_results()`

```
my_bool mysql_more_results(MYSQL *mysql)
```

Devuelve verdadero si existen más resultados en la consulta actualmente en ejecución, y la aplicación debe llamar a [mysql\\_next\\_result\(\)](#) para recuperar los resultados.

Esta función fue añadida en MySQL 4.1.0.

## Valores de retorno

*TRUE* (1) si existen más resultados. *FALSE* (0) si no existen.

En la mayoría de los casos, se puede llamar a [mysql\\_next\\_result\(\)](#) en lugar de comprobar si existen más resultados e iniciar una recuperación si es así.

## Errores

Ninguno.

# Función `mysql_next_result()`

```
int mysql_next_result(MYSQL *mysql)
```

Si existen más resultados de consultas, `mysql_next_result()` lee los siguientes resultados de consulta y devuelve el estado a la aplicación.

Se debe llamar a `mysql_free_result()` para la consulta anterior si devolvió un conjunto de resultados.

Después de llamar a `mysql_next_result()` el estado de la conexión es el mismo que si se hubiese llamado a `mysql_real_query()` o a `mysql_query()` para la siguiente consulta. Esto significa que se puede llamar a `mysql_store_result()`, `mysql_warning_count()`, `mysql_affected_rows()`, etc.

Si `mysql_next_result()` devuelve un error, no se ejecutará ninguna otra sentencia y no hay más resultados a recuperar.

Esta función se añadió en MySQL 4.1.0.

## Valores de retorno

Valor de retorno	Descripción
0	Éxito y hay más resultados
-1	Éxito y no hay más resultados
>0	Se ha producido un error

## Errores

`CR_COMMANDS_OUT_OF_SYNC`: Los comandos fueron ejecutados en un orden inapropiado.

`CR_SERVER_GONE_ERROR`: El servidor MySQL no está presente.

`CR_SERVER_LOST`: La conexión al servidor se perdió durante la consulta.

`CR_UNKNOWN_ERROR`: Se ha producido un error desconocido.

# Función `mysql_num_fields()`

```
unsigned int mysql_num_fields(MYSQL_RES *result);
```

O

```
unsigned int mysql_num_fields(MYSQL *mysql);
```

El segundo formato no funciona en la versión 3.22.24 de MySQL y siguientes. Para pasar un argumento **MYSQL\***, se debe usar en su lugar la función [mysql\\_field\\_count](#).

Devuelve el número de columnas en un conjunto de resultados.

También se puede obtener el número de columnas desde un puntero al conjunto de resultados o a un manipulador de conexión. Se puede usar el manipulador de conexión si [mysql\\_store\\_result](#) o [mysql\\_use\\_result](#) devuelve NULL (y por ese motivo no se dispone de un puntero al conjunto de resultados). En ese caso, se puede llamar a [mysql\\_field\\_count](#) para determinar si [mysql\\_store\\_result](#) ha producido un resultado no vacío. Esto permite al programa cliente tomar la acción apropiada sin saber si la consulta ha sido una sentencia **SELECT** (o del tipo *SELECT*). El ejemplo incluido cómo se puede realizar esto.

## Valores de retorno

Un entero sin signo que representa el número de campos en un conjunto de resultados.

## Errores

Ninguno.

## Ejemplo

```
MYSQL_RES *result;  
unsigned int num_fields;  
unsigned int num_rows;
```

```

if (mysql_query(&mysql,query_string)) {
    // error
}
else { // consulta exitosa, procesar cualquier dato devuelto por ella
    result = mysql_store_result(&mysql);
    if (result) { // no hay filas
        num_fields = mysql_num_fields(result);
        // recuperar filas, y después llamar a mysql_free_result(result)
    }
    else { // mysql_store_result() no devolvió nada; ¿debería?
        if (mysql_errno(&mysql)) {
            fprintf(stderr, "Error: %s\n", mysql_error(&mysql));
        }
        else if (mysql_field_count(&mysql) == 0) {
            // la consulta no ha devuelto datos
            // (no era un SELECT)
            num_rows = mysql_affected_rows(&mysql);
        }
    }
}
}
}

```

Una alternativa (si se sabe que la consulta debe devolver un conjunto de resultados) es reemplazar la llamada `mysql_errno(&mysql)` por una comprobación de si `mysql_field_count(&mysql)` es igual a 0. Esto sólo ocurrirá si algo ha salido mal.

## Función `mysql_num_rows()`

```
my_ulonglong mysql_num_rows(MYSQL_RES *result);
```

Devuelve el número de filas en el conjunto de resultados.

El uso de `mysql_num_rows` depende si se ha usado [mysql\\_store\\_result](#) o [mysql\\_use\\_result](#) para obtener el conjunto de resultados. Si se ha usado [mysql\\_store\\_result](#), `mysql_num_rows` puede ser llamada inmediatamente. Si se ha usado [mysql\\_use\\_result](#), `mysql_num_rows` no devuelve el valor correcto hasta que todas las filas del conjunto de resultados hayan sido recuperadas.

### Valor de retorno

El número de filas en el conjunto de resultados.

### Errores

Ninguno.

## Función `mysql_options()`

```
int mysql_options(MYSQL *mysql, enum mysql_option option, const char *arg)
```

Puede usarse para activar opciones de conexión extra y afectar al comportamiento de una conexión. Esta función puede ser llamada muchas veces para cambiar varias opciones.

`mysql_options()` debe ser llamada después de [mysql\\_init\(\)](#) y antes de [mysql\\_connect\(\)](#) o [mysql\\_real\\_connect\(\)](#).

El argumento *option* es la opción que se quiere activar; el argumento *arg* es el valor de la opción. Si la opción es un entero, entonces *arg* debe apuntar al valor del entero.

Posible valores de opciones:

Opción	Tipo de argumento	Función
MYSQL_INIT_COMMAND	char *	Comando a ejecutar cuando se conecte al servidor MySQL. Será reejecutado automáticamente cuando se reconecte.
MYSQL_OPT_COMPRESS	Not used	Usar el protocolo comprimido cliente/servidor.
MYSQL_OPT_CONNECT_TIMEOUT	unsigned int *	Tiempo límite para la conexión en segundos.
MYSQL_OPT_LOCAL_INFILE	puntero opcional a uint	Si no se proporciona un puntero o si el puntero apunta a un unsigned int != 0 el comando <a href="#">LOAD LOCAL INFILE</a> estará permitido.
MYSQL_OPT_NAMED_PIPE	No usado	Usar tuberías con nombre para conectar a un servidor MySQL en NT.
MYSQL_OPT_PROTOCOL	unsigned int *	Tipo de protocolo a usar. Debe ser uno de los valores enumerados <code>mysql_protocol_type</code> definido en 'mysql.h'. Nuevo en 4.1.0.

MYSQL_OPT_READ_TIMEOUT	unsigned int *	Tiempo límite para lecturas desde el servidor (actualmente sólo funciona en Windows con conexiones TCP/IP). Nuevo en 4.1.1.
MYSQL_OPT_WRITE_TIMEOUT	unsigned int *	Tiempo límite para escrituras en el servidor (actualmente sólo funciona en Windows con conexiones TCP/IP). Nuevo en 4.1.1.
MYSQL_READ_DEFAULT_FILE	char *	Lee las opciones desde el fichero nombrado en lugar de hacerlo desde 'my.cnf'.
MYSQL_READ_DEFAULT_GROUP	char *	Lee las opciones desde el grupo nombrado de 'my.cnf' o desde el fichero especificado con <b>MYSQL_READ_DEFAULT_FILE</b> .
MYSQL_REPORT_DATA_TRUNCATION	my_bool *	Activa o desactiva informes de errores de truncado de datos para sentencias preparadas vía <b>MYSQL_BIND.error</b> . (Por defecto: desactivada) Nuevo en 5.0.3.
MYSQL_SECURE_AUTH	my_bool*	Si para conectar al servidor no está soportado el nuevo tipo de contraseñas 4.1.1. Nuevo en 4.1.1.
MYSQL_SET_CHARSET_DIR	char*	El nombre de camino del directorio que contiene los ficheros de definición de juegos de caracteres.
MYSQL_SET_CHARSET_NAME	char*	El nombre del juego de caracteres a usar como juego de caracteres por defecto.
MYSQL_SHARED_MEMORY_BASE_NAME	char*	Nombre del objeto de memoria compartida para comunicarse con el servidor. Debe ser la misma que para la opción <b>-shared-memory-base-name</b> usada para el servidor <b>mysqld</b> al que se quiere conectar. Nuevo en 4.1.0.

Hay que tener en cuenta que el grupo del cliente siempre es leído si se usa **MYSQL\_READ\_DEFAULT\_FILE** o **MYSQL\_READ\_DEFAULT\_GROUP**.

El grupo especificado en el fichero de opciones debe contener las opciones siguientes:

Opción	Descripción
connect-timeout	Tiempo límite de conexión en segundos. En <b>Linux</b> este tiempo también se usa para esperar la primera respuesta del servidor.
compress	Usar el protocolo cliente/servidor comprimido.
database	Conectar a esta base de datos si no se especifica una en el comando de conexión.
debug	Opciones de depuración.
disable-local-infile	Deshabilitar el uso de <a href="#">LOAD DATA LOCAL</a> .
host	Nombre de ordenador por defecto.
init-command	Comando a ejecutar cuando se conecte al servidor MySQL. Será reejecutado automáticamente cuando se reconecte.
interactive-timeout	Lo mismo que especificar CLIENT_INTERACTIVE en <a href="#">mysql_real_connect()</a> .
local-infile[=(0 1)]	Si no hay argumento o si argumento != 0 se habilita el uso de <a href="#">LOAD DATA LOCAL</a> .
max_allowed_packet	Tamaño máximo de paquete que el cliente puede leer del servidor.
multi-results	Permite múltiples conjuntos de resultados desde ejecuciones de sentencias múltiples o procedimientos de almacenamiento. Nuevo en 4.1.1.
multi-statements	Permite al cliente enviar múltiples sentencias en una única cadena (separadas por ';'). Nuevo en 4.1.9.
password	Contraseña por defecto.
pipe	Usar tuberías con nombre para conectar al servidor MySQL en NT.
protocol={TCP   SOCKET   PIPE   MEMORY}	Protocolo a usar cuando se conecte al servidor (Nuevo en 4.1)
port	Número de puerto por defecto.
return-found-rows	Hace que <a href="#">mysql_info()</a> devuelva las filas encontradas en lugar de las actualizadas cuando se usa <a href="#">UPDATE</a> .

shared-memory-base-name=name	Nombre de memoria compartida a usar para conectar al servidor (por defecto es "MYSQL"). Nuevo en MySQL 4.1.
socket	Fichero <i>socket</i> por defecto.
user	Usuario por defecto.

Nótese que *timeout* ha sido remplazado por *connect-timeout*, pero *timeout* seguirá funcionando durante todavía.

## Valores de retorno

Cero si tiene éxito. Distinto de cero si se usa una opción desconocida.

## Ejemplo

```

MYSQL mysql;

(&mysql);
mysql_options(&mysql,MYSQL_OPT_COMPRESS,0);
mysql_options(&mysql,MYSQL_READ_DEFAULT_GROUP,"odbc");
if (!mysql_real_connect(&mysql,"host","user","passwd","database",0,
NULL,0))
{
    fprintf(stderr,"Error al conectar a la base de datos: Error: %s\n",
        mysql_error(&mysql));
}

```

Este código pide al cliente que use el protocolo cliente/servidor comprimido y lee las opciones adicionales desde la sección *odbc* del fichero 'my.cnf'.

# Función `mysql_ping()`

```
int mysql_ping(MYSQL *mysql)
```

Comprueba si la conexión con el servidor está funcionando. Si la conexión ha caído, se intenta una reconexión automática.

Esta función puede ser usada por clientes que permanecen inactivas por mucho tiempo, para comprobar si el servidor ha cerrado la conexión y reconectarla si es necesario.

## Valores de retorno

Cero si el servidor responde. Distinto de cero si ha ocurrido un error.

## Errores

CR\_COMMANDS\_OUT\_OF\_SYNC: Los comandos fueron ejecutados en un orden inapropiado.

CR\_SERVER\_GONE\_ERROR: El servidor MySQL no está presente.

CR\_UNKNOWN\_ERROR: Se ha producido un error desconocido.

# Función `mysql_query()`

```
int mysql_query(MYSQL *mysql, const char *query);
```

Ejecuta una consulta SQL apuntada por la cadena terminada con cero del parámetro *query*. La consulta debe consistir en una sentencia SQL simple. No se debe añadir el punto y coma al final (;) o \g a la cadena. Si la ejecución de múltiples sentencias está permitida, la cadena puede contener varias sentencias separadas por punto y coma.

**mysql\_query** no puede ser usado para consultas que contengan datos binarios; en esos casos se debe usar [mysql\\_real\\_query](#). (Los datos binarios pueden contener el carácter '\0', que **mysql\_query** interpreta como el final de la cadena de consulta.)

Si se quiere averiguar si la consulta devuelve un conjunto de resultados o no, se puede usar [mysql\\_field\\_count](#) para verificarlo.

## Valor de retorno

El valor de retorno es cero si la consulta se ha completado correctamente. Un valor distinto de cero indica que ha ocurrido un error.

## Errores

CR\_COMMANDS\_OUT\_OF\_SYNC: los comandos fueron ejecutados en un orden inapropiado.

CR\_SERVER\_GONE\_ERROR: el servidor MySQL no está presente.

CR\_SERVER\_LOST: la conexión con el servidor se perdió durante la consulta.

CR\_UNKNOWN\_ERROR: ha ocurrido un error desconocido.

## Función `mysql_real_connect()`

```
MYSQL *mysql_real_connect(MYSQL *mysql, const char *host, const char
*usuario, const char *password,
    const char *database, unsigned int puerto, const char *unix_socket,
unsigned long client_flag);
```

**mysql\_real\_connect** intenta establecer una conexión con un motor de bases de datos MySQL ejecutándose en *host*. **mysql\_real\_connect** debe completarse con éxito antes de que se puedan ejecutar otras funciones del API, con la excepción de [mysql\\_get\\_client\\_info](#).

Los parámetros a especificar son los siguientes:

- **mysql**: El primer parámetro debe ser la dirección de una estructura [MYSQL](#) existente. Antes de llamar a **mysql\_real\_connect** se debe llamar a [mysql\\_init](#) para inicializar la estructura [MYSQL](#). Se pueden modificar muchas opciones de conexión mediante la función [mysql\\_options](#).
- **host**: El valor de *host* puede ser tanto un nombre como una dirección IP. Si *host* es NULL o la cadena "localhost", se asume que se trata de una conexión al host local. Si el sistema operativo soporta sockets (Unix) o tuberías con nombre (Windows), se en lugar del protocolo TCP/IP para conectar con el servidor.
- **usuario**: contiene el identificador de login del usuario MySQL. Si el user es NULL o una cadena vacía "", se asume el usuario actual. Bajo Unix, es el nombre de login actual. Bajo Windows ODBC, el nombre de usuario debe especificarse explícitamente.
- **password** contiene el password del usuario. Si es NULL, sólo se verificarán usuarios de la tabla que tengan como password una cadena vacía. Esto permite al administrador de la base de datos configurar el sistema de privilegios de MySQL de modo que existan usuarios con diferentes privilegios dependiendo del password especificado. Nota: no se debe intentar encriptar el password antes de llamar a **mysql\_real\_connect**; la encriptación del password se hace automáticamente por el API del cliente.
- **database** es el nombre de la base de datos. Si *database* no es NULL, la conexión hará que esa sea la base de datos por defecto.
- **puerto** si no es 0, el valor se usará como número de puerto en la conexión TCP/IP. Hay que tener en cuenta que el parámetro *host* determina el tipo de conexión.
- **unix\_socket** si no es NULL, la cadena especifica el socket o tubería con nombre que se usará. Hay que tener en cuenta que el parámetro *host* determina el tipo de conexión.
- **client\_flag** es normalmente 0, pero puede usarse una combinación de los siguientes flags en circunstancias muy especiales:

Flag	Descripción
------	-------------

CLIENT_COMPRESS	Usar un protocolo de compresión.
CLIENT_FOUND_ROWS	Devuelve el número de líneas encontradas (coincidentes) en lugar del número de líneas afectadas.
CLIENT_IGNORE_SPACE	Permite espacios después de los nombres de función. Hace de todos los nombres de función palabras reservadas.
CLIENT_INTERACTIVE	Permite segundos "interactive_timeout" (en lugar de segundos "wait_timeout") de inactividad antes de cerrar la conexión.
CLIENT_LOCAL_FILES	Permite manipulación <a href="#">LOAD DATA LOCAL</a> .
CLIENT_MULTI_STATEMENTS	Informa al servidor de que el cliente puede enviar consultas multilínea (separadas con `;`). Si este flag no se activa, las consultas multilínea serán desactivadas. Nuevo en versión 4.1.
CLIENT_MULTI_RESULTS	Informa al servidor de que el cliente puede manipular conjuntos de resultados multiples procedentes de multi-consultas o procedimientos almacenados. Esto es agrupado automáticamente si CLIENT_MULTI_STATEMENTS es activado. Nuevo en 4.1.
CLIENT_NO_SCHEMA	No permite la sintaxis <i>db_name.tbl_name.col_name</i> . Esto es para <b>ODBC</b> . Hace que el analizador sintáctico genere un error si se usa tal sintaxis, lo que resulta útil para encontrar errores en algunos programas <b>ODBC</b> .
CLIENT_ODBC	El cliente es un cliente ODBC. Esto hace que mysqlqld se adapte más a ODBC.
CLIENT_SSL	Usar SSL (protocolo de encriptado). Esta opción no debe ser activada por aplicaciones; se activa internamente por la librería del cliente.

## Valores de retorno

Si la conexión ha tenido éxito, el valor de retorno es un manipulador de conexión [MYSQL\\*](#), NULL si la conexión no ha tenido éxito. Para una conexión exitosa, el valor de retorno es el mismo que el valor del primer parámetro.

## Errores

CR\_CONN\_HOST\_ERROR: fallo al conectar con el servidor MySQL.

CR\_CONNECTION\_ERROR: fallo al conectar al servidor local MySQL.

CR\_IPSOCK\_ERROR: fallo al crear un socket IP.

CR\_OUT\_OF\_MEMORY: falta memoria.

CR\_SOCKET\_CREATE\_ERROR: fallo al crear un socket Unix.

CR\_UNKNOWN\_HOST: fallo al encontrar una dirección IP para el nombre de host.

CR\_VERSION\_ERROR: al intentar conectar con el servidor se ha producido un error de protocolo por el uso de una librería de cliente que usa un protocolo de una versión diferente. Esto puede ocurrir si se usa una librería de cliente muy antigua para conectarse con un servidor que no se ha arrancado con la opción "--old-protocol".

CR\_NAMEDPIPEOPEN\_ERROR: fallo al crear una tubería con nombre en Windows.

CR\_NAMEDPIPEWAIT\_ERROR: fallo al esperar uan tubería con nombre en Windows.

CR\_NAMEDPIPESETSTATE\_ERROR: fallo al tomar un manipulador de tubería en Windows.

CR\_SERVER\_LOST: si el connect\_timeout > 0 y se tarda más de connect\_timeout segundos en conectar con el servidor o si el servidor ha muerto mientras se ejecuta el comando de inicialización.

## Ejemplo

```
MYSQL mysql;

(&mysql);
(&mysql, MYSQL_READ_DEFAULT_GROUP, "your_prog_name");
if (!mysql_real_connect(&mysql, "host", "user", "passwd", "database", 0,
NULL, 0))
{
    fprintf(stderr, "Fallo al conectar con la base de datos: Error: %s\n",
        mysql_error(&mysql));
}
```

Mediante el uso de [mysql\\_options\(\)](#) la librería MySQL leerá las secciones [client] y [your\_prog\_name] del fichero 'my.cnf', lo que asegura que el programa funcionará, aunque alguien haya arrancado MySQL de algún modo no estándar.

Notar que una vez conectado, **mysql\_real\_connect()** activa la opción de reconexión (parte de la estructura [MYSQL](#)) a un valor de 1 en versiones del API anteriores a 5.0.3, y de 0 en versiones más recientes. Un valor 1 para esta opción indica, en el caso de que una consulta no pueda ser completada por una pérdida de conexión, se intente reconectar al servidor antes de abandonar.

## Función `mysql_real_escape_string()`

```
unsigned long mysql_real_escape_string(MYSQL *mysql, char *to, const char
*from, unsigned long length)
```

El parámetro *mysql* debe ser una conexión abierta válida. Esto es necesario ya que el escapado depende del conjunto de caracteres en uso por el servidor.

Esta función se usa para crear una cadena SQL legal que se puede usar en una sentencia SQL.

La cadena en *from* se codifica como una cadena SQL escapada, teniendo en cuenta el conjunto de caracteres actual de la conexión. El resultado se coloca en *to* y se añade un carácter nulo terminador. Los caracteres codificados son NUL (ASCII 0), '\n', '\r', '\', '"', '''' y Control-Z. (Estrictamente hablando, MySQL sólo requiere que se escapen los caracteres de barra invertida y el carácter de comilla usado para entrecomillar la cadena. Esta función entrecomilla los otros caracteres para que sean más fáciles de leer en ficheros de diario.)

La cadena apuntada por *from* debe tener *long* bytes de longitud. Además, se debe crear un buffer *to* con al menos *length\*2+1* bytes de longitud. (En el peor caso, cada carácter necesitará ser codificado usando dos bytes, y se necesita espacio para el carácter terminador nulo.) Cuando `mysql_real_escape_string()` regresa, el contenido de *to* será una cadena terminada en nulo. El valor de retorno es la longitud de la cadena codificada, sin incluir el carácter nulo terminador.

### Ejemplo

```
char query[1000],*end;

end = strmov(query,"INSERT INTO test_table values(");
*end++ = '\\';
end += mysql_real_escape_string(&mysql, end,"Qué es esto",11);
*end++ = '\\';
*end++ = ',';
*end++ = '\\';
end += mysql_real_escape_string(&mysql, end,"datos binarios: \\0\\r\\n",19);
*end++ = '\\';
*end++ = ')';

if (mysql_real_query(&mysql,query,(unsigned int) (end - query)))
{
```

```
fprintf(stderr, "Fallo al insertar fila, Error: %s\n",  
         mysql_error(&mysql));  
}
```

La función **strmov()** usada en el ejemplo está incluida en la librería `mysqlclient` y trabaja igual que **strcpy()** pero devuelve un puntero al terminador nulo del primer parámetros.

## Valores de retorno

La longitud del valor colocado en *to*, sin incluir el carácter nulo terminador.

## Errores

Ninguno.

# Función `mysql_real_query()`

```
int mysql_real_query(MYSQL *mysql, const char *query, unsigned long length)
```

Ejecuta la consulta SQL apuntada por *query*, que debe ser una cadena de *length* bytes de longitud. Normalmente, la cadena debe consistir en una sentencia SQL simple y no se debe añadir el punto y coma (;) terminador o \g a la sentencia. Si la ejecución de múltiples sentencias está permitida, la cadena puede contener varias sentencias separadas con punto y coma.

Se debe usar `mysql_real_query()` en lugar de `mysql_query()` para consultas que contengan datos binarios, porque los datos binarios pueden contener el carácter '\0'. Además, `mysql_real_query()` es más rápido que `mysql_query()` porque no llama a `strlen()` para la cadena *query*.

Si se quiere saber si la consulta debe devolver un conjunto de resultados, se puede usar `mysql_field_count()` para comprobarlo.

## Valores de retorno

Cero si la consulta tuvo éxito. Distinto de cero si se produjo algún error.

## Errores

CR\_COMMANDS\_OUT\_OF\_SYNC: Los comandos fueron ejecutados en un orden inapropiado.

CR\_SERVER\_GONE\_ERROR: El servidor MySQL no está presente.

CR\_SERVER\_LOST: La conexión al servidor se perdió durante la consulta.

CR\_UNKNOWN\_ERROR: Se ha producido un error desconocido.

# Función `mysql_reload()`

```
int mysql_reload(MYSQL *mysql)
```

Obliga al servidor MySQL a recargar las tablas de privilegios. El usuario conectado debe tener el privilegio *RELOAD*.

Esta función está desaconsejada. Es preferible usar [mysql\\_query\(\)](#) para lanzar una sentencia SQL [FLUSH PRIVILEGES](#) en su lugar.

## Valores de retorno

Cero si tiene éxito. Distinto de cero si se produjo un error.

## Errores

CR\_COMMANDS\_OUT\_OF\_SYNC: Los comandos fueron ejecutados en un orden inapropiado.

CR\_SERVER\_GONE\_ERROR: El servidor MySQL no está presente.

CR\_SERVER\_LOST: La conexión al servidor se perdió durante la consulta.

CR\_UNKNOWN\_ERROR: Se ha producido un error desconocido.

## Tipo MYSQL\_RES

```
typedef struct st_mysql_res {
    my_ulonglong row_count;
    MYSQL_FIELD  *fields;
    MYSQL_DATA   *data;
    MYSQL_ROWS   *data_cursor;
    unsigned long *lengths;           /* column lengths of current row */
    MYSQL        *handle;             /* for unbuffered reads */
    MEM_ROOT     field_alloc;
    unsigned int  field_count, current_field;
    MYSQL_ROW    row;                 /* If unbuffered read */
    MYSQL_ROW    current_row;         /* buffer to current row */
    my_bool      eof;                 /* Used by mysql_fetch_row */
    /* mysql_stmt_close() had to cancel this result */
    my_bool      unbuffered_fetch_cancelled;
    const struct st_mysql_methods *methods;
} MYSQL_RES;
```

## ***Función mysql\_rollback()***

```
my_bool mysql_rollback(MYSQL *mysql)
```

Vuelva atrás la transacción actual.

Esta función se añadió en MySQL 4.1.0.

### **Valores de retorno**

Cero si tiene éxito. Distinto de cero si ocurre algún error.

### **Errores**

Ninguno.

## *Tipo MYSQL\_ROW*

```
typedef char **MYSQL_ROW;           /* return data as array of strings
*/
```

## ***Tipo MYSQL\_ROWS***

```
typedef struct st_mysql_rows {
    struct st_mysql_rows *next;           /* list of rows */
    MYSQL_ROW data;
    unsigned long length;
} MYSQL_ROWS;
```

# Función `mysql_row_seek()`

```
MYSQL_ROW_OFFSET mysql_row_seek(MYSQL_RES *result, MYSQL_ROW_OFFSET offset)
```

Coloca el cursor de fila en una fila arbitraria dentro de un conjunto de resultados de una consulta. El valor *offset* es un desplazamiento que debe ser devuelto por [mysql\\_row\\_tell\(\)](#) o por [mysql\\_row\\_seek\(\)](#). Este valor no es un número de fila; si se quiere situar en una fila dentro de un conjunto de resultados mediante un número, usar [mysql\\_data\\_seek\(\)](#) en su lugar.

Esta función requiere que la estructura del conjunto de resultados contenga el resultado completo de una consulta, de modo que `mysql_row_seek()` sólo puede ser usado en conjunción con [mysql\\_store\\_result\(\)](#), y no con [mysql\\_use\\_result\(\)](#).

## Valores de retorno

El valor previo del cursor de fila. Este valor puede ser pasado a una llamada sucesiva a `mysql_row_seek()`.

## Errores

Ninguno.

# Función `mysql_row_tell()`

```
MYSQL_ROW_OFFSET mysql_row_tell(MYSQL_RES *result)
```

Devuelve la posición actual del cursor de fila para la última llamada a [mysql\\_fetch\\_row\(\)](#). Este valor puede ser usado como argumento en una llamada a [mysql\\_row\\_seek\(\)](#).

Se debe usar **mysql\_row\_tell()** sólo después de una llamada a [mysql\\_store\\_result\(\)](#), no después de [mysql\\_use\\_result\(\)](#).

## Valores de retorno

El desplazamiento actual del cursor de fila.

## Errores

Ninguno.

## Función `mysql_select_db()`

```
int mysql_select_db(MYSQL *mysql, const char *database);
```

Hace que la base de datos *database* especificada se convierta en la base de datos por defecto (actual) en la conexión especificada mediante *mysql*. En consultas posteriores, esta base de datos será la usada para referencias a tablas que no incluyan un especificador de base de datos.

**mysql\_select\_db** falla a no ser que el usuario conectado pueda ser autenticado como poseedor de permisos para usar la base de datos.

### Valor de retorno

Cero si la operación ha tenido éxito, otro valor si no es así.

### Errores

CR\_COMMANDS\_OUT\_OF\_SYNC: los comandos fueron ejecutados en un orden inapropiado.

CR\_SERVER\_GONE\_ERROR: el servidor MySQL no está presente.

CR\_SERVER\_LOST: la conexión con el servidor se ha perdido durante la consulta.

CR\_UNKNOWN\_ERROR: ha ocurrido un error desconocido.

Otros errores:

ER\_BAD\_DB\_ERROR: la base de datos especificada no existe.

# Función `mysql_set_server_option()`

```
int mysql_set_server_option(MYSQL *mysql, enum enum_mysql_set_option option)
```

Activa o desactiva una opción para la conexión *mysql*. *option* puede ser uno de los siguientes valores:

<code>MYSQL_OPTION_MULTI_STATEMENTS_ON</code>	Activar soporte para sentencias múltiples.
<code>MYSQL_OPTION_MULTI_STATEMENTS_OFF</code>	Desactivar soporte para sentencias múltiples.

Esta función fue añadida en MySQL 4.1.1.

## Valores de retorno

Cero si tiene éxito, distinto de cero si se produjo un error.

## Errores

`CR_COMMANDS_OUT_OF_SYNC`: Los comandos fueron ejecutados en un orden inapropiado.

`CR_SERVER_GONE_ERROR`: El servidor MySQL no está presente.

`CR_SERVER_LOST`: La conexión al servidor se perdió durante la consulta.

`CR_UNKNOWN_COM_ERROR`: El servidor no soporta **`mysql_set_server_option()`** (este es el caso cuando el servidor es anterior a 4.1.1) o el servidor no soporta la opción que se intenta activar.

# Función `mysql_shutdown()`

```
int mysql_shutdown(MYSQL *mysql, enum enum_shutdown_level shutdown_level)
```

Indica al servidor de bases de datos que se apague. El usuario conectado debe tener el privilegio `SHUTDOWN`. El argumento de `shutdown_level` se añadió en MySQL 4.1.3 (y 5.0.1). El servidor MySQL actualmente soporta sólo un tipo (nivel de cortesía) de apagado; `shutdown_level` debe ser igual a `SHUTDOWN_DEFAULT`. Más adelante se añadirán más niveles y entonces el argumento `shutdown_level` permitirá elegir el nivel deseado. Los servidores MySQL y clientes MySQL anteriores y posteriores a 4.1.3 son compatibles; los servidores MySQL más recientes que 4.1.3 aceptan llamadas `mysql_shutdown(MYSQL *mysql)`, y servidores MySQL anteriores a 4.1.3 aceptan la nueva llamada `mysql_shutdown()`. Pero los ejecutables enlazados dinámicamente que hayan sido compilados con versiones de cabeceras antiguas de `libmysqlclient`, y que llamen a `mysql_shutdown()`, tienen que ser usados con la vieja librería dinámica `libmysqlclient`.

## Valores de retorno

Cero si tiene éxito, distinto de cero si se produjo un error.

## Errores

`CR_COMMANDS_OUT_OF_SYNC`: Los comandos fueron ejecutados en un orden inapropiado.

`CR_SERVER_GONE_ERROR`: El servidor MySQL no está presente.

`CR_SERVER_LOST`: La conexión al servidor se perdió durante la consulta.

`CR_UNKNOWN_ERROR`: Se ha producido un error desconocido.

# Función `mysql_sqlstate()`

```
const char *mysql_sqlstate(MYSQL *mysql)
```

Devuelve una cadena terminada en null que contiene el código de error SQLSTATE para el último error. El código de error consiste en cinco caracteres. '00000' significa "sin error". Los valores están especificados por ANSI SQL y ODBC.

Hay que tener en cuenta que no todos los errores de MySQL han sido ya mapeados a errores SQLSTATE. El valor 'HY000' (error general) se usa para errores sin mapear.

Esta función fue añadida en MySQL 4.1.1.

## Valores de retorno

Una cadena terminada en nulo que contiene el código de error SQLSTATE.

## Ver también

[mysql\\_errno\(\)](#), [mysql\\_error\(\)](#) y [mysql\\_stmt\\_sqlstate\(\)](#).

## Función `mysql_ssl_set()`

```
int mysql_ssl_set(MYSQL *mysql, const char *key,  
                 const char *cert, const char *ca, const char *capath, const char *cipher)
```

`mysql_ssl_set()` se usa para establecer conexiones seguras usando SSL. Debe ser llamada antes de `mysql_real_connect()`.

`mysql_ssl_set()` no hace nada a no ser que el soporte OpenSSL esté activo en la librería del cliente.

`mysql` es el manipulador de conexión devuelto por `mysql_init()`. El resto de los parámetros se especifican como sigue:

- `key` es el camino para el fichero llave.
- `cert` es el camino para el fichero de certificado.
- `ca` es el camino para el fichero de certificado de autoridad.
- `capath` es el camino al directorio que contiene los certificados de confianza SSL CA en formato pem.
- `cipher` es una lista de claves disponibles para usar para encriptado SSL.

Cualquier parámetro SSL sin usar debe ser dado como NULL.

### Valores de retorno

Esta función siempre devuelve 0. Si el sistema SSL es incorrecto, `mysql_real_connect()` devolverá un error cuando se intente la conexión.

# Función `mysql_stat()`

```
char *mysql_stat(MYSQL *mysql)
```

Devuelve una cadena de caracteres que contiene información similar a la proporcionada por el comando `mysqladmin status`. Esto incluye el tiempo activo en segundos y el número de procesos en ejecución, preguntas, recargas y tablas abiertas.

## Valores de retorno

Una cadena de caracteres que describe el estado del servidor. NULL si se produce un error.

## Errores

CR\_COMMANDS\_OUT\_OF\_SYNC: Los comandos fueron ejecutados en un orden inapropiado.

CR\_SERVER\_GONE\_ERROR: El servidor MySQL no está presente.

CR\_SERVER\_LOST: La conexión al servidor se perdió durante la consulta.

CR\_UNKNOWN\_ERROR: Se ha producido un error desconocido.

## Función `mysql_store_result()`

```
MYSQL_RES *mysql_store_result(MYSQL *mysql);
```

Se debe usar `mysql_store_result()` o `mysql_use_result()` para cada consulta que haya recuperado datos (`SELECT`, `SHOW`, `DESCRIBE`, `EXPLAIN`, `CHECK TABLE`, etc).

No es necesario llamarlas para otras consultas, pero no harán ningún daño ni resultará una pérdida apreciable si se llama a `mysql_store_result()` en todos los casos. Se puede comprobar si la consulta no ha proporcionado resultados comprobando si `mysql_store_result()` devuelve cero.

Si se quiere saber si una consulta debe devolver un conjunto de resultados o no, se puede usar `mysql_field_count()`.

`mysql_store_result()` lee el resultado total de una consulta al cliente, asigna una estructura `MYSQL_RES` y coloca el resultado en esa estructura.

`mysql_store_result()` devuelve un puntero nulo si la consulta no ha devuelto un conjunto de resultados (si la consulta ha sido, por ejemplo, una sentencia `INSERT`).

`mysql_store_result()` también devuelve un puntero nulo si la lectura del conjunto de resultados falla. Se puede verificar si se trata de un error comprobando si `mysql_error()` devuelve una cadena no vacía, si `mysql_errno()` devuelve un valor distinto de cero o si `mysql_field_count()` devuelve un valor cero.

Se obtiene un conjunto de resultados vacío si no hay filas devueltas. (No es lo mismo un conjunto de resultados vacío que un puntero nulo como valor de retorno.)

Una vez que se ha llamado a `mysql_store_result()` y se ha obtenido un resultado que no sea un puntero nulo, se puede llamar a `mysql_num_rows()` para encontrar cuántas filas hay en el conjunto de resultados.

También se puede llamar a `mysql_fetch_row()` para recuperar filas desde el conjunto de resultados, o `mysql_row_seek()` y `mysql_row_tell()` para obtener o modificar la posición actual de la fila dentro del conjunto de resultados.

Se debe llamar a `mysql_free_result()` una vez que se ha terminado de procesar el conjunto de resultados.

A veces `mysql_store_result()` devuelve NULL después aunque `mysql_query()` regrese con éxito.

- Se ha producido un error al reservar memoria dinámica, por ejemplo, si el conjunto de resultados es demasiado grande.
- Los datos no pudieron ser leídos, si ha ocurrido un error durante la conexión.
- La consulta no ha devuelto datos, por ejemplo, una consulta de tipo INSERT, UPDATE o DELETE).

## Valores de retorno

Una estructura MYSQL\_RES con los resultados o NULL si se ha producido un error.

## Errores

**mysql\_store\_result()** resetea mysql\_error() y mysql\_errno() si tiene éxito.

**CR\_COMMANDS\_OUT\_OF\_SYNC**: los comandos fueron ejecutados en un orden inapropiado.

**CR\_OUT\_OF\_MEMORY**: no hay memoria.

**CR\_SERVER\_GONE\_ERROR**: el servidor MySQL no está presente.

**CR\_SERVER\_LOST**: la conexión con el servidor se perdió durante la consulta.

**CR\_UNKNOWN\_ERROR**: ha ocurrido un error desconocido.

## Función `mysql_thread_id()`

```
unsigned long mysql_thread_id(MYSQL *mysql)
```

Devuelve el ID del proceso para la conexión actual. Este valor puede ser usado como argumento en la función [mysql\\_kill\(\)](#) para matar el proceso.

Si la conexión se pierde y se reconecta con [mysql\\_ping\(\)](#), el ID del proceso cambiará. Esto significa que no se debe leer el ID del proceso y almacenarlo para usarlo más tarde. Se debe obtener en el momento en que se necesite.

### Valores de retorno

El ID del proceso de la conexión actual.

### Errores

Ninguno.

# Función `mysql_use_result()`

```
MYSQL_RES *mysql_use_result(MYSQL *mysql)
```

Se debe usar [mysql\\_store\\_result\(\)](#) o `mysql_use_result()` para cada consulta que haya recuperado datos ([SELECT](#), [SHOW](#), [DESCRIBE](#), [EXPLAIN](#), [CHECK TABLE](#), etc).

`mysql_use_result()` inicia una recuperación de un conjunto de resultados, pero no lee el conjunto de resultados en el cliente, como hace [mysql\\_store\\_result\(\)](#). En su lugar, cada fila debe ser recuperada individualmente mediante llamadas a [mysql\\_fetch\\_row\(\)](#). Esto lee el resultado de una consulta directamente desde el servidor sin almacenarlo en una tabla temporal o en un buffer local, lo que es algo más rápido y requiere mucha menos memoria que [mysql\\_store\\_result\(\)](#). El cliente reservará memoria sólo para la fila actual y un buffer de comunicación que puede crecer hasta un máximo de `max_allowed_packet` bytes.

Por otra parte, no se debe usar `mysql_use_result()` si se va a hacer mucho proceso para cada fila en el lado del cliente, o si la salida se envía a una pantalla en la que el usuario puede usar ^S (parar el desplazamiento). Esto bloqueará el servidor e impide que otros procesos puedan actualizar las tablas para las que los datos están siendo recuperados.

Cuando se usa `mysql_use_result()`, se debe ejecutar [mysql\\_fetch\\_row\(\)](#) hasta que devuelva un valor NULL, en caso contrario, las filas no recuperadas se devolverán como parte del conjunto de resultados de la siguiente consulta. El API C obtendrá un error "Commands out of sync; you can't run this command now if you forget to do this!".

No se debe usar [mysql\\_data\\_seek\(\)](#), [mysql\\_row\\_seek\(\)](#), [mysql\\_row\\_tell\(\)](#), [mysql\\_num\\_rows\(\)](#) o [mysql\\_affected\\_rows\(\)](#) con un resultado devuelto desde `mysql_use_result()`, tampoco se deben lanzar otras consultas hasta que `mysql_use_result()` haya finalizado. (Sin embargo, después de que se hayan recuperado todas las filas, [mysql\\_num\\_rows\(\)](#) devolverá con precisión el número de filas recuperadas.)

Se debe llamar a [mysql\\_free\\_result\(\)](#) una vez que se haya terminado de procesar el conjunto de resultados.

## Valores de retorno

Una estructura de resultado [MYSQL\\_RES](#). NULL si se produce un error.

## Errores

**mysql\_use\_result()** resetea [mysql\\_error](#) y [mysql\\_errno](#) si tiene éxito.

**CR\_COMMANDS\_OUT\_OF\_SYNC:** los comandos fueron ejecutados en un orden inapropiado.

**CR\_OUT\_OF\_MEMORY:** no hay memoria.

**CR\_SERVER\_GONE\_ERROR:** el servidor MySQL no está presente.

**CR\_SERVER\_LOST:** la conexión con el servidor se perdió durante la consulta.

**CR\_UNKNOWN\_ERROR:** ha ocurrido un error desconocido.

## Función `mysql_warning_count()`

```
unsigned int mysql_warning_count(MYSQL *mysql)
```

Devuelve el número de avisos generados durante la ejecución de la sentencia SQL previa.

Esta función fue añadida en MySQL 4.1.0.

### Valores de retorno

El contador de avisos.

### Errores

Ninguno.