CURSO DE MySQL

Prof. Lic. Jomar R. Gómez Robles, Mastère-Telecom

Tabla de contenido.

__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.

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, extrictamente 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, miestras 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 asociaciación 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 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 interelaciones 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 propidad 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 si 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 péstamo, 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.

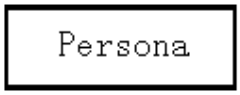
En realidad, es irrelevante si una entidad en fruto de una generalización o de una especialización, no deja de ser una entidad, y por lo tanto, no afecta al modelo.

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:



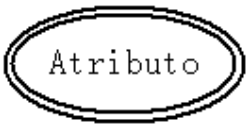
Las entidades débiles pueden representarse mediante dos rectángulos inscritos. Ya sabemos que existe una dependencia de existencia entre la entidad débil y la fuerte, esto se representa también añadiendo una flecha a la línea que llega a la entidad débil.

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:

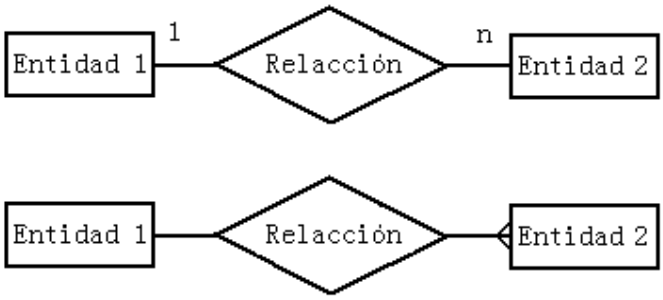


Atributo multivaluado: (o multivalorado) se dice del atributo tal que para una misma entidad puede tomar varios valores diferentes, es decir, varios valores del mismo dominio.

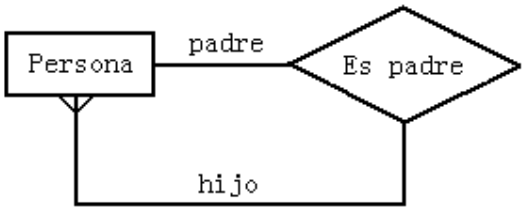
Interrelación

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 intervienten 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:

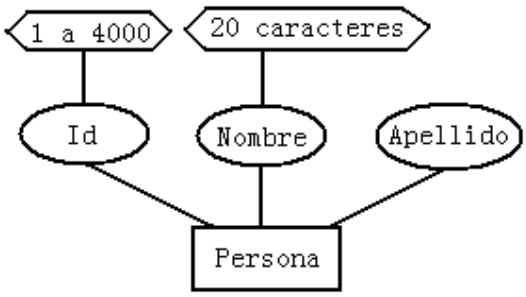


Sobre las líneas a veces se añade el rol que representa cada entidad:



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 necesitemos. 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.

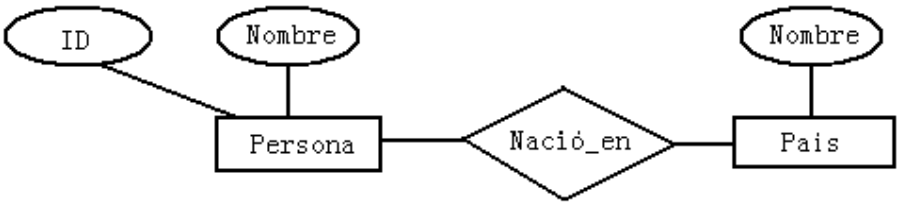
Construir un modelo E-R

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.

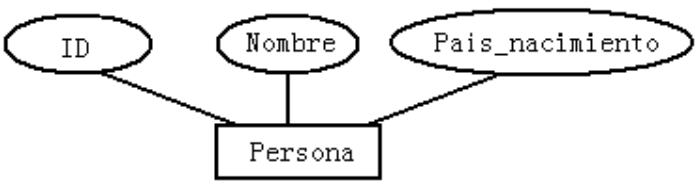
Uno de los primeros problemas con que nos encontraremos será decidir qué son entidades y qué atributos.

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?.

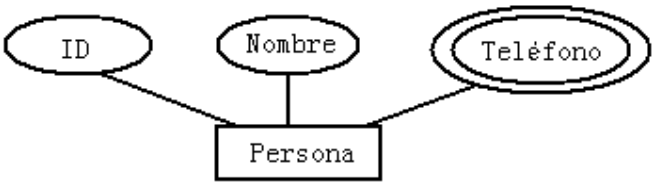


Una regla que puede ayudarnos en esta decisión es que si una entidad sólo tiene un atributo, que sirve para identificarlo, entonces esa entidad puede ser considerara como un atributo.

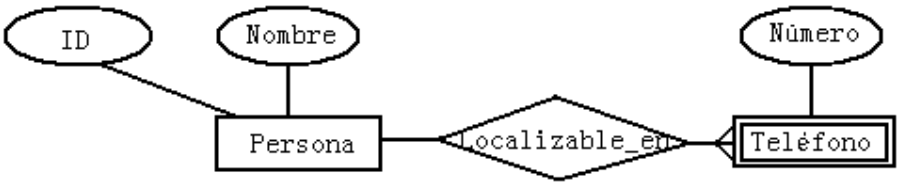


Otro problema frecuente se presenta con los atributos multivaluados.

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 diagráma 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 elegido 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.

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 amplia 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 puden 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 cadinalidad 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 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 interreación 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:

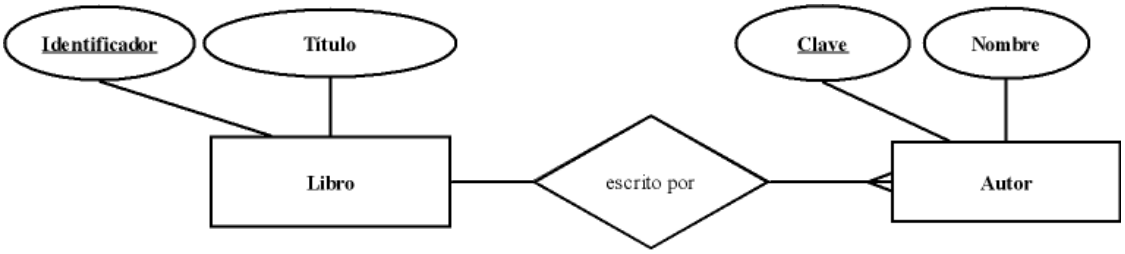
1. Para cada conjunto de entidades fuertes se crea una relación con una columna para cada atributo.
2. Para cada conjunto de entidades débiles se crea una relación que contiene una columna para los atributos que forman la clave primaria de la entidad fuerte a la que se encuentra subordinada y una columna 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 relacionar 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)
tabla
id   nombre   apellido    fecha       estado
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 Gionzialez  2/6/1970    viudo
```

Algunos ejemplos de selección serían:

```
tabla[id<'200']
id   nombre   apellido    fecha       estado
123  Fulano   Prierez     4/12/1987   soltero
102  Tulana   Liopez      24/6/1985   casado
005  Tutulano Gionzialez  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

id	nombre	apellido	fecha	estado
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	Gionzialez	2/6/1970	viudo

Algunos ejemplos de proyección serían:

tabla[id,apellido]

id	apellido
123	Prierez
454	Sianchiez
102	Liopez
554	Gaomez
005	Gionzialez

tabla[nombre, estado]

nombre	estado
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.

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 atribito 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)
```

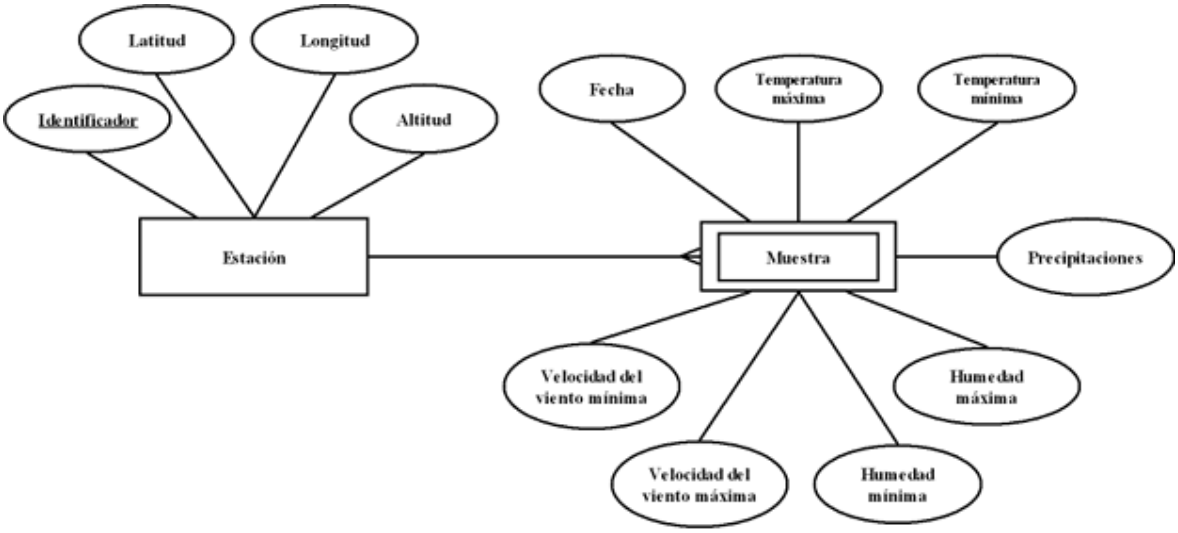
A esto se le denomina propagación de la clave de la entidad *Editorial* a la entidad *Libro*.

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:

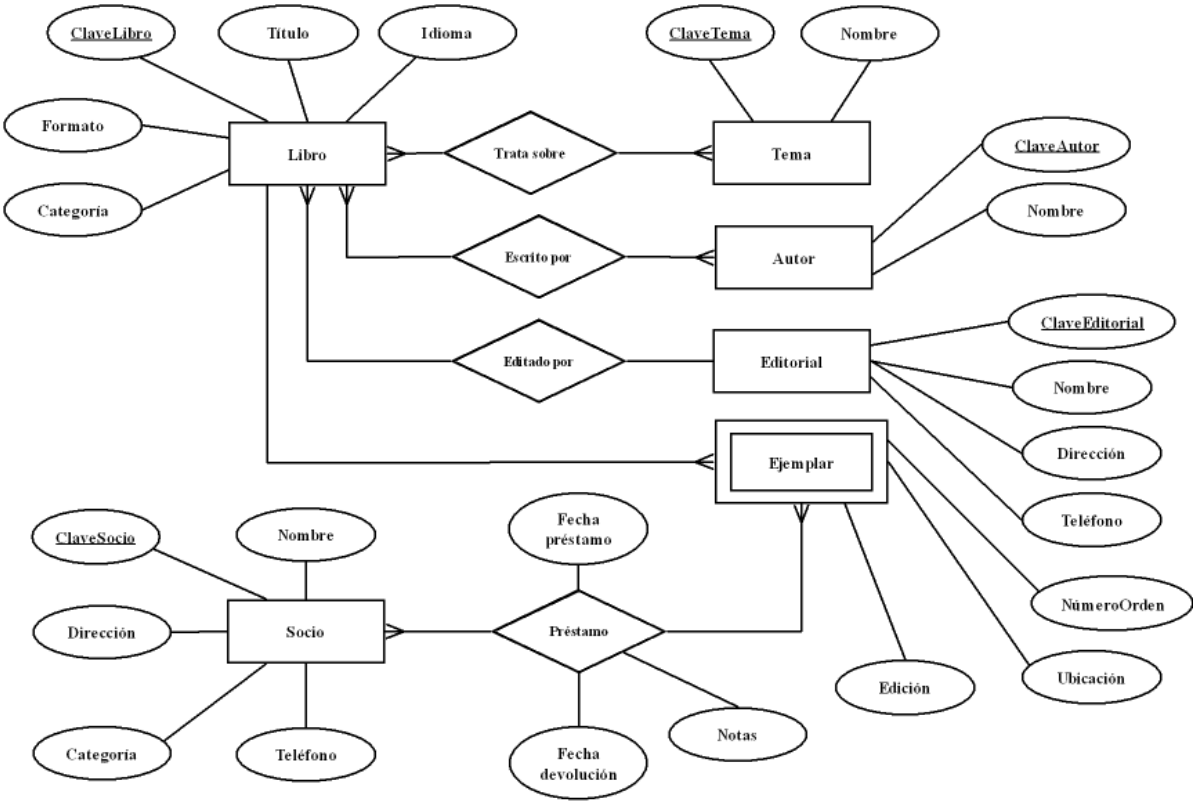


Sólo necesitamos dos tablas para hacer la conversión:

```
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 módelos 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 inconsistencas. 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 una atributo con un identificador único de préstamo.

Como vemos, las restrinciones 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 pié 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
Titulo                autor     fecha       editorial
```

```
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 almancenados 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 -> Y`.

El símbolo `->` 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 `|` significa negación. Así `X -> |Y` se lee como *X no determina Y*.

Dependencia funcional completa

Definición: En una dependencia funcional `X -> 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 => Y`.

Dependecia funcional elemental

Definición: Si tenemos una dependencia completa `X => 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.

Dependecia funcional trivial

Definición: Una dependencia funcional `A -> 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 -> Y`, `Y -> Z`, `Y -> |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 -> país`, `país -> continente`. Además, `país -> |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:

Ciudades

ID_ciudad	Nombre	población	superficie	renta	país	continente
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)
Paises(nombre_pais**(PK)**, nombre_continente)

Ciudades

ID_ciudad	Nombre	población	superficie	renta	país
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

Paises

país	continente
Francia	Europa
Alemania	Europa
China	Asia

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

Forma Normal de Boycce 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 ->-> 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*: `nombre ->-> 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:

- `nombre -> fecha_nacimiento`
- `nombre -> estado_civil`
- `nombre ->-> teléfono`, o lo que es lo mismo `(nombre,teléfono) -> 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:

Agenda

nombre	fecha_nacimiento	estado_civil	teléfono
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 separandolos 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

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

Teléfonos

nombre	teléfono
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 multivaludada 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

nombre	fecha_nacimiento	estado_civil	teléfono	correo
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

nombre	fecha_nacimiento	estado_civil	teléfono	correo
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

nombre	fecha_nacimiento	estado_civil	teléfono	correo
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

nombre	fecha_nacimiento	estado_civil	teléfono	correo
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 ->-> Y` no trivial, X es clave candidata.

Una dependencia multivaluada `A ->-> 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 ->-> teléfono`
- `nombre ->-> 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 defienen 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 singo, 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 singo, 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 singo, 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 **FLOAT(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 qu 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).

LONGBLOB
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 'N90º00'00.000"' y 'S90º00'00.000"'. 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 título, 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 paises 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.

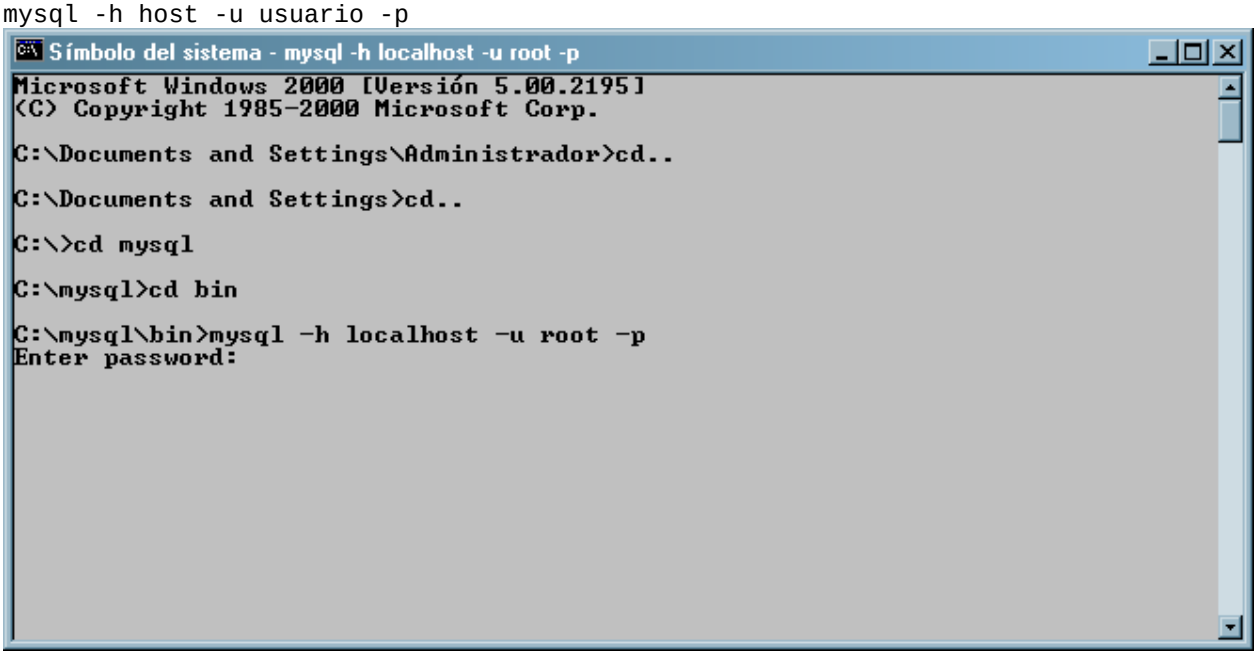
Para este curso usaremos **MySQL** de forma directa, mediante un cliente ejecutándose en una consola (una ventana DOS en Windows, o un Shell en otros sistemas). En otras secciones se explicarán los diferentes APIs.

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
```



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 Manipilation 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. Deberemos indicar el nombre de la tabla y los nombres y tipos de las columnas:

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

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 autoinrementada 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

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

telefonos3

numero	id
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

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

telefonos3

numero	id
12322132	1
12332221	1
55546545	**4**
55565445	**4**

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 opiones 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, creemos 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>
```

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 númericas 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 permitem, 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 consitía en seleccionar determinados atributos de una relación.

Mediante la sentencia SELECT es posible hacer una proyección de una tabla, seleccionando las columas 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 consitía 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 clásula *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 porteriormente en expresiones.

Para crear una variable hay dos posibilidades. La primera consiste en ulsar 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, require 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 lo 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 mabos 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 <=>, 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 = 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 !=. 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 considerá 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 si 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);
+----------------------------+
| GREATEST(2,5,"7",1,"a",12) |
+----------------------------+
|                         12 |
+----------------------------+
1 row in set (0.09 sec)

mysql>
```

Verificar conjuntos

Los operadores *IN* y *NOT IN* sirven para averiguar si el valor de una expresión determinada está dentro de un conjunto indicado. La sintaxis es:

```
 IN (<expr1>, <expr2>, <expr3>...)
 NOT IN (<expr1>, <expr2>, <expr3>...)
```

El operador *IN* devuelve un valor verdadero, 1, si el valor de la expresión es igual a alguno de los valores especificados en la lista. El operador *NOT IN* devuelve un valor falso en el mismo caso. Por ejemplo:

```
mysql> SELECT 10 IN(2, 4, 6, 8, 10);
+-----------------------+
| 10 IN(2, 4, 6, 8, 10) |
+-----------------------+
|                     1 |
+-----------------------+
1 row in set (0.00 sec)

mysql>
```

Verificar nulos

El operador *ISNULL* es equivalente a *IS NULL*.

La sintaxis es:

```
ISNULL(<expresión>)
```

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 la 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 consegir 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 izquieda 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 izquieda 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*$_i$ 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
LOG	Logaritmo en base arbitraria
LOG10	Logaritmo en base 10
LOG2	Logaritmo en base dos
MOD o %	Resto de una división entera
PI	Valor del número π
POW o POWER	Valor de potencias
RADIANS	Conversión de radianes a grados
RAND	Valores aleatorios
ROUND	Cálculo de redondeos
SIGN	Devuelve el signo
SIN	Cálculo del seno de un ángulo
SQRT	Cálculo de la raíz cuadrada

TAN	Cálculo de la tangente de un ángulo
TRUNCATE	Elimina decimales

Funciones de cadenas

Las funciones para tratamiento de cadenas de caracteres son:

ASCII	Valor de código ASCII de un carácter
BIN	Converión a binario
BIT_LENGTH	Cálculo de longitud de cadena en bits
CHAR	Convierte de ASCII a carácter
CHAR_LENGTH o CHARACTER_LENGTH	Cálculo de longitud de cadena en caracteres
COMPRESS	Comprime una cadena de caracteres
CONCAT	Concatena dos cadenas de caracteres
CONCAT_WS	Concatena cadenas con separadores
CONV	Convierte números entre distintas bases
ELT	Elección entre varias cadenas
EXPORT_SET	Expresiones binarias como conjuntos
FIELD	Busca el índice en listas de cadenas
FIND_IN_SET	Búsqueda en listas de cadenas
HEX	Conversión de números a hexadecimal
INSERT	Inserta una cadena en otra
INSTR	Busca una cadena en otra
LEFT	Extraer parte izquierda de una cadena
LENGTH u OCTET_LENGTH	Calcula la longitud de una cadena en bytes
LOAD_FILE	Lee un fichero en una cadena
LOCATE o POSITION	Encontrar la posición de una cadena dentro de otra
LOWER o LCASE	Convierte una cadena a minúsculas
LPAD	Añade caracteres a la izquierda de una cadena
LTRIM	Elimina espacios a la izquierda de una cadena
MAKE_SET	Crea un conjunto a partir de una expresión binaria
OCT	Convierte un número a octal
ORD	Obtiene el código ASCII, incluso con caracteres multibyte
QUOTE	Entrecomilla una cadena
REPEAT	Construye una cadena como una repetición de otra
REPLACE	Busca una secuencia en una cadena y la sustituye por otra
REVERSE	Invierte el orden de los caracteres de una cadena

RIGHT	Devuelve la parte derecha de una cadena
RPAD	Inserta caracteres al final de una cadena
RTRIM	Elimina caracteres blancos a la derecha de una cadena
SOUNDEX	Devuelve la cadena "soundex" para una cadena concreta
SOUNDS LIKE	Compara cadenas según su pronunciación
SPACE	Devuelve cadenas consistentes en espacios
SUBSTRING o MID	Extraer subcadenas de una cadena
SUBSTRING_INDEX	Extraer subcadenas en función de delimitadores
TRIM	Elimina sufijos y/o prefijos de una cadena.
UCASE o UPPER	Convierte una cadena a mayúsculas
UNCOMPRESS	Descomprime una cadena comprimida mediante COMPRESS
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 encripdado

Funciones de encriptado de datos y de checksum:

AES_ENCRYPT y AES_DECRYPT	Encriptar y desencriptar datos usando el algoritmo oficial AES
DECODE	Desencripta una cadena usando una contraseña
ENCODE	Encripta una cadena usando una contraseña
DES_DECRYPT	Desencripta usando el algoritmo Triple-DES
DES_ENCRYPT	Encripta usando el algoritmo Triple-DES
ENCRYPT	Encripta str usando la llamada del sistema Unix *crypt()*
MD5	Calcula un checksum MD5 de 128 bits para la cadena string
PASSWORD u OLD_PASSWORD	Calcula una cadena contraseña a partir de la cadena en texto plano
SHA o SHA1	Calcula un checksum SHA1 de 160 bits para una cadena

Funciones de información

Información sobre el sistema:

BENCHMARK	Ejecuta una expresión varias veces
CHARSET	Devuelve el conjunto de caracteres de una cadena
COERCIBILITY	Devuelve el valor de restricción de colección de una cadena
COLLATION	Devuelve la colección para el conjunto de caracteres de una cadena
CONNECTION_ID	Devuelve el ID de una conexión
CURRENT_USER	Devuelve el nombre de usuario y el del host para la conexión actual
DATABASE	Devuelve el nombre de la base de datos actual
FOUND_ROWS	Calcular cuántas filas se hubiesen obtenido en una sentencia SELECT sin la cláusula LIMIT
LAST_INSERT_ID	Devuelve el último valor generado automáticamente para una columna AUTO_INCREMENT
USER o SESSION_USER o SYSTEM_USER	Devuelve el nombre de usuario y host actual de MySQL
VERSION	Devuelve la versión del servidor MySQL

Miscelanea

Funciones generales:

DEFAULT	Devuelve el valor por defecto para una columna
FORMAT	Formatea el número según la plantilla '#,###,###.##
GET_LOCK	Intenta obtener un bloqueo con el nombre dado

INET_ATON	Obtiene el entero equivalente a la dirección de red dada en formato de cuarteto con puntos
INET_NTOA	Obtiene la dirección en formato de cuarteto con puntos dado un entero
IS_FREE_LOCK	Verifica si un nombre de bloqueo está libre
IS_USED_LOCK	Verifica si un nombre de bloqueo está en uso
MASTER_POS_WAIT	Espera hasta que el esclavo alcanza la posición especificada en el diario maestro
RELEASE_LOCK	Libera un bloqueo
UUID	Devuelve un identificador único universal

De grupos

Funciones de grupos:

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

12 Lenguaje SQL
Consultas multitabla

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=telefonos2.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 eComposiciones 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 |
|  3 | Tulanita | 1985-03-17 | NULL     | NULL | (1)
|  4 | Fusganita | 1976-08-25 | 456545654 |    4 |
|  4 | Fusganita | 1976-08-25 | 441415414 |    4 |
+----+----------+------------+----------+------+
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>
```

legimos 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>
```

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 constraseña 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 constraseña 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 teminamos 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 sentecia:

```
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 sentecia 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>
```

SQL

Sentencias

ALTER TABLE
ANALYZE TABLE
BACKUP TABLE
BEGIN
BEGIN WORK
CHECK TABLE
CHECKSUM TABLE
COMMIT
CREATE DATABASE
CREATE TABLE
CREATE USER
DELETE
DESCRIBE
DO
DROP DATABASE
DROP INDEX
DROP TABLE
DROP USER
FLUSH GRANT
HANDLER
INSERT INSERT ... SELECT
INSERT DELAYED
JOIN
KILL
LOAD DATA
LOCK TABLES
OPTIMIZE TABLE
RENAME TABLE
REPAIR TABLE
REPLACE
RESET
REVOKE
ROLLBACK
SELECT

Indice de Funciones SQL (194)

BENCHMARK BIN
BIT_AND BIT_LENGTH
BIT_OR BIT_XOR

-C- 

CAST CEIL
CEILING CHAR
CHARACTER_LENGTH CHARSET
CHAR_LENGTH COERCIBILITY
COLLATION COMPRESS
CONCAT CONCAT_WS
CONNECTION_ID CONV
CONVERT CONVERT_TZ
COS COT
COUNT COUNT DISTINCT
CRC32 CURDATE
CURRENT_DATE CURRENT_TIME
CURRENT_TIMESTAMP CURRENT_USER
CURTIME

-D- 

DATABASE DATE
DATEDIFF DATE_ADD
DATE_FORMAT DATE_SUB
DAY DAYNAME
DAYOFMONTH DAYOFWEEK
DAYOFYEAR DECODE
DEFAULT DEGREES
DES_DECRYPT DES_ENCRYPT
DIV

-E- 

ELT ENCODE
ENCRYPT EXP
EXPORT_SET EXTRACT

-F- 

FIELD FIND_IN_SET
FLOOR FORMAT
FOUND_ROWS FROM_DAYS
FROM_UNIXTIME

-G- 

-H- 

-I- 

-L- 

-M- 

-N- 

-O- 

-P- 

PASSWORD PERIOD_ADD
PERIOD_DIFF PI
POSITION POW
POWER

-Q-

QUARTER QUOTE

-R-

RADIANS RAND
RELEASE_LOCK REPEAT
REPLACE REVERSE
RIGHT ROUND
RPAD RTRIM

-S-

SECOND SEC_TO_TIME
SESSION_USER SHA
SHA1 SIGN
SIN SOUNDEX
SOUNDS_LIKE SPACE
SQRT STD
STDDEV STRCMP
STR_TO_DATE SUBDATE
SUBSTRING SUBSTRING_INDEX
SUBTIME SUM
SYSDATE SYSTEM_USER

-T-

TAN TIME
TIMEDIFF TIMESTAMP
TIMESTAMPADD TIMESTAMPDIFF
TIME_FORMAT TIME_TO_SEC
TO_DAYS TRIM
TRUNCATE

-U-

UCASE UNCOMPRESS
UNCOMPRESSED_LENGTH UNHEX
UNIX_TIMESTAMP UPPER
USER UTC_DATE

Indice de Funciones y tipos API C SQL (72)

En mayúsculas aparecen los tipos definidos en **MySQL**, en minúsculas, las funciones del API C.

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.

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 uncluir caracteres multi-byte, hay que tener en cuenta que las longitudes máximas mostradas en la tabla se cuentan en bytes. Si un identificadore 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 d eun 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 signigica 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 nos 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 refiera 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 CREATE TABLE o CREATE DATABASE. Las comparaciones de nombres son sensibles al tipo. Este es el valor en sistemas **Unix**. Si se fuerza este valor 0 con --lower-case-table-names=0

	en un sistema de ficheros no sensible al tipo y se accede a nombres de tablas **MyISAM** usando diferentes tipos de caracteres, se puede producir corrupción de índices.
1	Los bombres de las tablas se almacenan en disco usando minúsculas y las comparaciones de nombres no son sensibles al tipo. **MySQL** 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 **Windows** y **Mac OS X**.
2	Los nombres de tablas y bases de datos se almacenan en disco usando el tipo de letra especificado en las sentencias CREATE TABLE o CREATE DATABASE, pero **MySQL** 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 **InnoDB** se almacenan en minúscylas, igual que si lower_case_table_names=1. Asignar lower_case_table_names a 2 se puede hacer desde MySQL 4.0.18.

Si sólo se está usando **MySQL** para una plataforma, generalmente no será necerasio modificar la variable lower_case_table_names variable. 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 dene 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 impementación 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 contrucciones:

^

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 culaquier 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 'pipi' 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 carácteres que coincide con todos los caracteres pertenecientes a esa clase. Los nombres de clases estándar son:

alnum	Caracteres alfanuméricos
alpha	Caracteres alfabéticos
blank	Carácteres 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 cara´cter 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 iniciale es 'SYSTEM', que indica que el huso horario del servidor es wl 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 puedem 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 systema. 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 e 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 subdirectirio '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 la 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
LONGBLOB	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 tutotiales 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

- **Curso Tomado de MySQL con Clase**

http://mysql.conclase.net/curso/index.php?cap=000