Curso de programación en MySQL
(MySQL Versión 5)
Manual del alumno

Ing. Cedric Simon – Tel: 2268 0974 – Cel: 8888 2387 – Email: cedric@solucionjava.com – Web: www.solucionjava.com

SolucionJava.com

1 Índice

2 Introducción al curso

2.1 Objetivo de este curso

En este curso vamos a aprender como programar disparadores, procedimientos, y funciones en MySQL versión 5.0.27.

2.2 Manual del alumno

Este manual del alumno es una ayuda para el alumno, para tenga un recuerdo del curso. Este manual contiene un resumen de las materias que se van a estudiar durante el curso, pero el alumno debería de tomar notas personales para completas este manual.

En el CD del curso viene la documentación de MySQL en español, que completa este manual. La mayoria de los puntos de este manual están explicado de manera más completa en la documantación de MySQL en el CD.

2.3 Ejercicios prácticos

Para captar mejor la teoría, se harán muchos ejercicios con los alumnos, para probar la teoría y verificar la integración de la materia.

También, el alumno podrá copiar sus códigos en un disquete al fin del curso para llevarse, con fin de seguir la práctica en su hogar.

2.4 Requisitos para atender a este curso

Se requiere un conocimiento de base del lenguaje SQL.

Si el alumno tiene dificultades en un u otro capitulo, el debe sentirse libre de pedir explicaciones adicionales al profesor.

2.5 Soporte despues del curso

Si tienes preguntas sobre la materia del curso en tus ejercicios prácticos, puedes escribir tus preguntas a cedric@solucionjava.com.

Para informaciones sobre otros cursos, visita el sitio web www.solucionjava.com.

3 Genralidades de MySQL

3.1 Historia de MySQL

Empezamos con la intención de usar mSQL para conectar a nuestras tablas utilizando nuestras propias rutinas rápidas de bajo nivel (ISAM). Sin embargo y tras algunas pruebas, llegamos a la conclusión que mSQL no era lo suficientemente rápido o flexible para nuestras necesidades. Esto provocó la creación de una nueva interfície SQL para nuestra base de datos pero casi con la misma interfície API que mSQL. Esta API fue diseñada para permitir código de terceras partes que fue escrito para poder usarse con mSQL para ser fácilmente portado para el uso con MySQL.

La derivación del nombre MySQL no está clara. Nuestro directorio base y un gran número de nuestras bibliotecas y herramientas han tenido el prefijo "my" por más de 10 años. Sin embargo, la hija del co-fundador Monty Widenius también se llama My. Cuál de los dos dió su nombre a MySQL todavía es un misterio, incluso para nosotros.

El nombre del delfín de MySQL (nuestro logo) es "Sakila", que fué elegido por los fundadores de MySQL AB de una gran lista de nombres sugerida por los usuarios en el concurso "Name the Dolphin" (ponle nombre al delfín). El nombre ganador fue enviado por Ambrose Twebaze, un desarrollador de software Open Source de Swaziland, África. Según Ambrose, el nombre femenino de Sakila tiene sus raíces en SiSwate, el idioma local de Swaziland. Sakila también es el nombre de una ciudad en Arusha, Tanzania, cerca del país de origen de Ambrose, Uganda.

3.2 Las principales características de MySQL

La siguiente lista describe algunas de las características más importantes del software de base de datos MySQL.

3.2.1 Interioridades y portabilidad

- Escrito en C y en C++
- Probado con un amplio rango de compiladores diferentes
- Funciona en diferentes plataformas.
- Usa GNU Automake, Autoconf, y Libtool para portabilidad.
- APIs disponibles para C, C++, Eiffel, Java, Perl, PHP, Python, Ruby, y Tcl.
- Uso completo de multi-threaded mediante threads del kernel. Pueden usarse fácilmente multiple CPUs si están disponibles.
- Proporciona sistemas de almacenamiento transaccionales y no transaccionales.
- Usa tablas en disco B-tree (MyISAM) muy rápidas con compresión de índice.
- Relativamente sencillo de añadir otro sistema de almacenamiento. Esto es útil si desea añadir una interfície SQL para una base de datos propia.
- Un sistema de reserva de memoria muy rápido basado en threads.
- Joins muy rápidos usando un multi-join de un paso optimizado.
- Tablas hash en memoria, que son usadas como tablas temporales.
- Las funciones SQL están implementadas usando una librería altamente optimizada y deben ser tan rápidas como sea posible. Normalmente no hay reserva de memoria tras toda la inicialización para consultas.
- El código MySQL se prueba con Purify (un detector de memoria perdida comercial) así como con Valgrind, una herramienta GPL (http://developer.kde.org/~sewardj/).
- El servidor está disponible como un programa separado para usar en un entorno de red cliente/servidor. También está disponible como biblioteca y puede ser incrustado (linkado) en

aplicaciones autónomas. Dichas aplicaciones pueden usarse por sí mismas o en entornos donde no hay red disponible..

3.2.2 Tipos de columnas

Diversos tipos de columnas: enteros con/sin signo de 1, 2, 3, 4, y 8 bytes de longitud, FLOAT, DOUBLE, CHAR, VARCHAR, TEXT, BLOB, DATE, TIME, DATETIME, TIMESTAMP, YEAR, SET, ENUM, y tipos espaciales OpenGIS.

3.2.3 Seguridad

Un sistema de privilegios y contraseñas que es muy flexible y seguro, y que permite verficación basada en el host. Las contraseñas son seguras porque todo el tráfico de contraseñas está encriptado cuando se conecta con un servidor.

3.2.4 Escalabilidad y límites

Soporte a grandes bases de datos. Usamos MySQL Server con bases de datos que contienen 50 millones de registros. También conocemos usuarios que usan MySQL Server con 60.000 tablas y acerca de 5.000.000 de registros.

Se permiten hasta 64 índices por tabla (32 antes de MySQL 4.1.2). Cada índice puede consistir desde 1 hasta 16 columnas o partes de columnas. El máximo ancho de límite son 1000 bytes (500 antes de MySQL 4.1.2).Un índice puede usar prefijos de una columna para los tipos de columna CHAR, VARCHAR, BLOB, o TEXT.

3.2.5 Conectividad

Los clientes pueden conectar con el servidor MySQL usando sockets TCP/IP en cualquier plataforma. En sistemas Windows de la familia NT (NT,2000,XP, o 2003), los clientes pueden usar named pipes para la conexión. En sistemas Unix, los clientes pueden conectar usando ficheros socket Unix.

En MySQL 5.0, los servidores Windows soportan conexiones con memoria compartida si se inicializan con la opción --shared-memory. Los clientes pueden conectar a través de memoria compartida usando la opción --protocol=memory.

La interfaz para el conector ODBC (MyODBC) proporciona a MySQL soporte para programas clientes que usen conexiones ODBC (Open Database Connectivity). Por ejemplo, puede usar MS Access para conectar al servidor MySQL. Los clientes pueden ejecutarse en Windows o Unix. El código fuente de MyODBC está disponible. Todas las funciones para ODBC 2.5 están soportadas, así como muchas otras.

La interfaz para el conector J MySQL proporciona soporte para clientes Java que usen conexiones JDBC. Estos clientes pueden ejecutarse en Windows o Unix. El código fuente para el conector J está disponible.

3.2.6 Localización

El servidor puede proporcionar mensajes de error a los clientes en muchos idomas.

Soporte completo para distintos conjuntos de carácteres, incluyendo latin1 (ISO-8859-1), german, big5, ujis, y más. Por ejemplo, los carácteres escandinavos 'â', 'ä' y 'ö' están permitidos en nombres de tablas y columnas. El soporte para Unicode está disponible.

Todos los datos se guardan en el conjunto de carácteres elegido. Todas las comparaciones para columnas normales de cadenas de carácteres son case-insensitive.

La ordenación se realiza acorde al conjunto de carácteres elegido (usando colación Sueca por defecto).

3.2.7 Clientes y herramientas

MySQL server tiene soporte para comandos SQL para chequear, optimizar, y reparar tablas. Estos comandos están disponibles a través de la línea de comandos y el cliente mysqlcheck. MySQL también incluye myisamchk, una utilidad de línea de comandos muy rápida para efectuar estas operaciones en tablas MyISAM.

Todos los programas MySQL pueden invocarse con las opciones --help o -? para obtener asistencia en línea.

3.2.8 Estabilidad de MySQL

Esta sección trata las preguntas "¿Qué estabilidad tiene MySQL Server?" y, "¿Puedo fiarme de MySQL Server para este proyecto?" Intentaremos clarificar estas cuestiones y responder algunas preguntas importantes que preocupan a muchos usuarios potenciales. La información en esta sección se basa en datos recopilados de las listas de correo, que son muy activas para identificar problemas así como para reportar tipos de usos.

El código original se remonta a los principos de los años 80. En TcX, la predecesora de MySQL AB, el código MySQL ha funcionado en proyectos desde mediados de 1996 sin ningún problema. Cuando el software de base de datos MySQL fue distribuído entre un público más amplio, nuestros nuevos usuarios rápidamente encontraron trozos de código no probados. Cada nueva versión desde entonces ha tenido pocos problemas de portabilidad incluso considerando que cada nueva versión ha tenido muchas nuevas funcionalidades.

Cada versión de MySQL Server ha sido usable. Los problemas han ocurrido únicamente cuando los usuarios han probado código de las "zonas grises". Naturalmente, los nuevos usuarios no conocen cuáles son estas zonas; esta sección, por lo tanto, trata de documentar dichas áreas conocidas a día de hoy. Las descripciones mayormente se corresponden con la versión 3.23, 4.0 y 4.1 de MySQL Server. Todos los bugs reportados y conocidos se arreglan en la última versión, con las excepciones listadas en las secciones de bugs y que están relacionados con problemas de diseño.

3.2.9 Modulos de MySQL

El diseño de MySQL Server es multi capa, con módulos independientes. Algunos de los últimos módulos se listan a continuación con una indicación de lo bien testeados que están:

1.1.1.1 Replicatión (Estable)

Hay grandes grupos de servidores usando replicación en producción, con buenos resultados. Se trabaja para mejorar características de replicación en MySQL 5.x.

1.1.1.2 InnoDB tablas (Estable)

El motor de almacenamiento transaccional InnoDB es estable y usado en grandes sistemas de producción con alta carga de trabajo.

1.1.1.3 BDB tablas (Estable)

El código Berkeley DB es muy estable, pero todavía lo estamos mejorando con el interfaz del motor de almacenamiento transaccional BDB en MySQL Server.

1.1.1.4 Búsquedas Full-text (Estable)

Búsquedas Full-text es ámpliamente usada.

1.1.1.5 MyODBC 3.51 (Estable)

MyODBC 3.51 usa ODBC SDK 3.51 y es usado en sistemas de producción ámpliamente. Algunas cuestiones surgidas parecen ser cuestión de las aplicaciones que lo usan e independientes del controlador ODBC o la base de datos subyacente.

3.2.10 Dimensiones máximas de las tablas MySQL

En MySQL 5.0, usando el motor de almacenamiento MyISAM, el máximo tamaño de las tablas es de 65536 terabytes (256 ^ 7 - 1 bytes). Por lo tanto, el tamaño efectivo máximo para las bases de datos en MySQL usualmente los determinan los límites de tamaño de ficheros del sistema operativo, y no por límites internos de MySQL.

El motor de almacenamiento InnoDB mantiene las tablas en un espacio que puede ser creado a partir de varios ficheros. Esto permite que una tabla supere el tamaño máximo individual de un fichero. Este espacio puede incluir particiones de disco, lo que permite tablas extremadamente grandes. El tamaño máximo del espacio de tablas es 64TB.

4 Instalación a base de datos

4.1 Instalación de MySQL

En este curso vamos a utilizar la base de datos MySQL, en su versión 5.0.27. Por eso la vamos a instalar ya.

4.1.1 Bajo Linux

Para instalar MySQL vamos primero a entrar como el usuario Root (o usar su).

Luego abrimos una ventana de consola, introducemos el CD del curso, y vamos a instalar la version de MySQL que estaen el CD lanzando desde el CD la instrucción:
`rpm -iv MySQL-server-5.0.27-0.glibc23.i386.rpm` para instalar el sevidor
`rpm -iv MySQL-client-5.0.27-0.glibc23.i386.rpm` para instalar el cliente

Eso installo MySQL bajo /usr/bin.

Vamos a crear una carpeta /mysql conteniendo los atajos hacia programas de MySQL.
`. createMySQLlinks.sh`

Vamos ahora a cambiar la clave del usuario root. Para cambiar la clave, entra en /mysql y ejecuta : `/usr/bin/mysqladmin -u root password 'SolPHP'`. La nueva clave sera 'SolPHP'.

Para verificar que MySQL esta bien instalado y se inicia, ejecuta `'rcmysql restart'` como Root.

Y ahora vamos a crear la base de datos del curso:
```
cd /media/CD
/mysql/mysql -u root -pSolJava
create database curso;
exit;
/mysql/mysql -u root -pSolJava curso < curso.sql
```

4.1.2 Bajo Windows

Bajo Windows solo se corre el archivo mysql-5.0.27-win32.exe con un usuario con derecho de administrador.

Se requiere un sistema operativo Windows de 32 bits, tal como 9x, Me, NT, 2000, XP, o Windows Server 2003.

Se recomienda fuertemente el uso de un sistema operativo Windows basado en NT (NT, 2000, XP, 2003) puesto que éstos permiten ejecutar el servidor MySQL como un servicio

4.2 Instalación herramienta de desarrollo

Como herramienta de desarrollo vamos a utilizar la versión gratis de EMS MySQL Manager Lite, que corre bajo Windows. Hasta ahora no he encontrado herramienta gratis bajo Linux que permite gestionar comodamente las principales nuevas funciones de MySQL 5 que son los triggers, procedimientos, funciones, y vistas.

Existen muchas herramientas libres para MySQ, funccionando bajo Linux como Windows. Las mas conocidas son phpMyAdmin y MySQL GUI tools de MySQL AB. Pero ninguno de estas herramientas permite manejar de manera comoda las vistas, los triggers, o los procedimientos/funciones.

5 Motores de base de datos

5.1 MyISAM

Es el primer motor de MySQL. Por defecto se crean las tablas usando el motor MyISAM.

Este motor crea las tablas como archivos de texto. Un archivo por tabla.

El motor MyISAM es muy veloz, especialmente en lectura, pero no soporta (todavia) las claves primarias y secundarias, ni las transacciones (commit/rollback).

5.2 InnoDB

Es el motor para bases de datos transacionales y/o con claves primaria y secundaria. Es el motor que vamos a utilizar en este curso. Se puede definir como motor por defecto a nivel de la base de datos y/o a nivel del servidor.

5.3 Memory (HEAP)

Este motor crea tablas temporales en la memoria. El aceso es entonces muy rápido, pero consume memoria si se almacena muchas datos.

5.4 Otros

MySQL trae varios otros tipos de motores de bases de datos, con sus especificidades. Entre otros BDB, CSV,... Ver la documentación de MySQL para mas detalles.

6 Tipos de columas

6.1 *Tipos numéricos*

A continuación hay un resumen de los tipos de columnas numéricos.

M indica la anchura máxima para mostrar. La anchura máxima es 255.

Si especifica ZEROFILL para columnas numéricas,, MySQL añade automáticamente el atributo UNSIGNED en la columna.

SERIAL es un alias para BIGINT UNSIGNED NOT NULL AUTO_INCREMENT.

SERIAL DEFAULT VALUE en la definición de una columna de tipo entero es un alias para NOT NULL AUTO_INCREMENT UNIQUE.

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

- BIT[(*M*)]
  En un tipo de datos bit. M indica el número de bits por valor, de 1 a 64. El valor por defecto es 1 si se omite *M* .
- TINYINT[(M)] [UNSIGNED] [ZEROFILL]
  Un entero muy pequeño. El rango con signo es de -128 a 127. El rango sin signo es de 0 a 255.
- BOOL, BOOLEAN
  Son sinónimos para TINYINT(1). Un valor de cero se considera falso. Valores distintos a cero se consideran ciertos.
  En el futuro, se introducirá tratamiento completo de tipos booleanos según el estándard SQL.
- SMALLINT[(*M*)] [UNSIGNED] [ZEROFILL]
  Un entero pequeño. El rango con signo es de -32768 a 32767. El rango sin signo es de 0 a 65535.
- MEDIUMINT[(*M*)] [UNSIGNED] [ZEROFILL]
  Entero de tamaño medio. El rango con signo es de -8388608 a 8388607. El rango sin singo es de 0 a 16777215.

- INT[(*M*)] [UNSIGNED] [ZEROFILL]
  Un entero de tamaño normal. El rango con signo es de -2147483648 a 2147483647. El rango sin signo es de 0 a 4294967295.
- INTEGER[(M)] [UNSIGNED] [ZEROFILL]
  Es un sinónimo de INT.
- BIGINT[(M)] [UNSIGNED] [ZEROFILL]
  Un entero grande. El rango con signo es de -9223372036854775808 a 9223372036854775807. El rango sin signo es de 0 a 18446744073709551615.
  Algunos aspectos a considerar con respecto a las columnas BIGINT :
- FLOAT(p) [UNSIGNED] [ZEROFILL]
  Número con coma flotante. *p* representa la precisión. Puede ir de 0 a 24 para números de coma flotante de precisión sencilla y de 25 a 53 para números de coma flotante con doble precisión.
  Esta sintaxis se proprociona para compatibilidad con ODBC.
- FLOAT[(*M*,D)] [UNSIGNED] [ZEROFILL]
  Un número de coma flotante pequeño (de precisión simple). Los valores permitidos son de -3.402823466E+38 a -1.175494351E-38, 0, y de 1.175494351E-38 a 3.402823466E+38. FLOAT sin argumentos o FLOAT(*p*) (donde p está en el rango de 0 a 24) es un número de coma flotante con precisión simple.
- DOUBLE[(*M*,B)] [UNSIGNED] [ZEROFILL]
  Número de coma flotante de tamaño normal (precisión doble). Los valores permitidos son de -1.7976931348623157E+308 a -2.2250738585072014E-308, 0, y de 2.2250738585072014E-308 a 1.7976931348623157E+308. Un número de coma flotante con precision sencilla tiene una precisión de 7 decimales aproximadamente; un número con coma flotante de doble precisión tiene una precisión aproximada de 15 decimales.
- DOUBLE PRECISION[(M,D)] [UNSIGNED] [ZEROFILL], REAL[(M,*D*)] [UNSIGNED] [ZEROFILL]
- DECIMAL[(*M*[,D])] [UNSIGNED] [ZEROFILL]
  A partir de MySQL 5.0.3:
  Número de punto fijo exacto y empaquetado. *M* es el número total de dígitos y *D* es el número de decimales. El punto decimal y (para números negativos) el signo '-' no se tiene en cuenta en M. Si D es 0, los valores no tienen punto decimal o parte fraccional.
  Si se omite *D*, el valor por defecto es 0. Si se omite M, el valor por defecto es 10.
  Todos los cálculos básicos (+, -, *, /) con columnas DECIMAL se hacen con precisión de 64 dígitos decimales.
- DEC[(M[,D])] [UNSIGNED] [ZEROFILL], NUMERIC[(M[,D])] [UNSIGNED] [ZEROFILL], FIXED[(*M*[,*D*])] [UNSIGNED] [ZEROFILL]
  Son sinónimos para DECIMAL. El sinónimo FIXED está disponible por compatibilidad con otros

Toda la aritmética se hace usando valores BIGINT o DOUBLE, así que no debe usar enteros sin signos mayores que 9223372036854775807 (63 bits) except con funciones bit! Si lo hace, algunos de los últimos dígitos en el resultado pueden ser erróneos por culpa de errores de redondeo al convertir valores BIGINT a DOUBLE.

Para darles una idea, un INT unsigned permite almacenar un numero por segundo durante 135 años. un BIGINT unsigned, 1.000.000 de numeros por segundo durante mas de 500.000 años...

6.2 Fechas y horas

Un resumen de los tipos de columnas temporales se muestra a continuación.

- DATE

Una fecha. El rango soportado es de '1000-01-01' a '9999-12-31'. MySQL muestra valores DATE en formato 'YYYY-MM-DD', pero permite asignar valores a columnas DATE usando cadenas de carácteres o números.

- DATETIME
  Combinación de fecha y hora. El rango soportado es de '1000-01-01 00:00:00' a '9999-12-31 23:59:59'. MySQL muestra valores DATETIME en formato 'YYYY-MM-DD HH:MM:SS', pero permite asignar valores a las columnas DATETIME usando cadenas de carácteres o números.
- TIMESTAMP[(M)]
  Una marca temporal. El rango es de '1970-01-01 00:00:00' hasta el año 2037.
  Una columna TIMESTAMP es útil para registrar la fecha y hora de una operación INSERT o UPDATE . La primera columna TIMESTAMP en una tabla se rellena automáticamente con la fecha y hora de la operación más reciente si no le asigna un valor. Puede asignar a cualquier columna TIMESTAMP la fecha y hora actual asignándole un valor NULL .
  En MySQL 5.0, TIMESTAMP se retorna como una cadena de carácteres en el formato 'YYYY-MM-DD HH:MM:SS' cuya anchura de muestra son 19 carácteres. Si quiere obtener el valor como un número, debe añadir +0 a la columa timestamp .
- TIME
  Una hora. El rango es de '-838:59:59' a '838:59:59'. MySQL muestra los valores TIME en formato 'HH:MM:SS', pero permite asingar valores a columnas TIME usando números o cadenas de carácteres.
- YEAR[(2|4)]
  Un año en formato de dos o cuatro dígitos. El valor por defecto está en formato de cuatro dígitos. En formato de cuatro dígitos, los valores permitidos son de 1901 a 2155, y 0000. En formato de dos dígitos, los valores permitidos son de 70 a 69, representando los años de 1970 a 2069. MySQL muestra los valores YEAR en formato YYYY pero permite asignar valores a columnas YEAR usando cadenas de caráctere

Tipo de Columna	**“Cero” Valor**
DATETIME	'0000-00-00 00:00:00'
DATE	'0000-00-00'
TIMESTAMP	00000000000000
TIME	'00:00:00'
YEAR	0000

6.3 Cadenas de caracteres

Un resumen de los tipos de columnas de cadenas de carácteres se muestra a continuación.

En algunos casos, MySQL puede cambiar una columna de cadena de carácteres a un tipo diferente para un comando CREATE TABLE o ALTER TABLE .

Los tipos de cadenas de carácteres MySQL 5.0 incluyen algunas características que puede que no haya encontrado trabajando con versiones anteriores de MySQL anteriores a la 4.1:

- CHAR
  Es un sinónimo de CHAR(1).
- BINARY(M)
  El tipo BINARY es similar al tipo CHAR, pero almacena cadenas de datos binarios en lugar de cadenas de carácteres no binarias.

- VARBINARY(M)
  El tipo VARBINARY es similar al tipo VARCHAR, pero almacena cadenas de carácteres binarias en lugar de cadenas de carácteres no binarias.
- TINYBLOB
  Una columna BLOB con una longitud máxima de 255 (2^8 - 1) bytes.
- TINYTEXT
  Una columna TEXT con longitud máxima de 255 (2^8 - 1) carácteres.
- BLOB[(M)]
  Una columna BLOB con longitud máxima de 65,535 (2^16 - 1) bytes.
- TEXT[(M)]
  Una columna TEXT con longitud máxima de 65,535 (2^16 - 1) carácteres.
- MEDIUMBLOB
  Una columna BLOB con longitud de 16,777,215 (2^24 - 1) bytes.
- MEDIUMTEXT
  Una columna TEXT con longitud máxima de 16,777,215 (2^24 - 1) carácteres.
- LONGBLOB
  Una columna BLOB con longitud máxima de 4,294,967,295 o 4GB (2^32 - 1) bytes. La longitud máxima *efectiva* (permitida) de las columnas LONGBLOB depende del tamaño máximo configurado para los paquetes en el protocolo cliente/servidor y la memoria disponible.
- LONGTEXT
  Una columna TEXT con longitud máxima de 4,294,967,295 or 4GB (2^32 - 1) carácteres. La longitud máxima *efectiva* (permitida) de columnas LONGTEXT depende del tamaño máximo de paquete configurado en el protocolo cliente/servidor y la memoria disponible.
- ENUM('*value1*','value2',...)
  Una enumeración. Un objeto de cadena de carácteres que sólo puede tener un valor, elegido de una lista de valores '*value1*', 'value2', ..., NULL o el valor de error especial '' . Una columna ENUM puede tener un máximo de 65,535 valores distintos. Los valores ENUM se representan internamente como enteros.
- SET('*value1*','*value2*',...)
  Un conjunto. Un objeto de cadena de carácteres que puede tener cero o más valores que deben pertencer a la lista de valores '*value1*', 'value2', ... Una columna SET puede tener un máximo de 64 miembros. Los valores SET se representan internamente como enteros.

Los tipos CHAR y VARCHAR son similares, pero difieren en cómo se almacenan y recuperan. Desde MySQL 5.0.3, también difieren en la longitud máxima y en cómo se tratan los espacios finales.

Los tipos CHAR y VARCHAR se declaran con una longitud que indica el máximo número de carácteres que quiere almacenar. Por ejemplo, CHAR(30) puede almacenar hasta 30 carácteres.

La longitud de una columna CHAR se fija a la longitud que se declara al crear la tabla. La longitud puede ser cualquier valor de 0 a 255. Cuando los valores CHAR se almacenan, se añaden espacios a la derecha hasta las longitud específica. Cuando los valores CHAR se recuperan, estos espacios se borran. Los valores en columnas VARCHAR son cadenas de carácteres de longitud variable. En MySQL 5.0, la longitud puede especficarse de 0 a 255 antes de MySQL 5.0.3, y de 0 a 65,535 en 5.0.3 y versiones posteriores. (La máxima longitud efectiva de un VARCHAR en MySQL 5.0 se determina por el tamaño de registro máximo y el conjunto de carácteres usados. La longitud máxima total es de 65,532 bytes.)

En contraste con CHAR, VARCHAR almacena los valores usando sólo los carácteres necesarios, más un byte adicional para la longitud (dos bytes para columnas que se declaran con una longitud superior a 255).

6.4 Requisitos de almacenamiento según el tipo de columna

Los requerimientos de almacenamiento para cada uno de los tipos de columnas soportados por MySQL se listan por categoría.

El máximo tamaño de un registro en una tabla `MyISAM` es 65,534 bytes. Cada columna `BLOB` y `TEXT` cuenta sólo de cinco a nueve bytes más alla de su tamaño.

Si una tabla `MyISAM` incluye cualquier tipo de columna de tamaño variable, el formato de rebistro también tiene longitud variable. Cuando se crea una tabla. MySQL puede, bajo ciertas condiciones, cambiar una columna de tamaño variable a fijo o viceversa.

.

Requerimientos de almacenamiento para tipos numéricos

Tipo de columna	Almacenamiento requerido
`TINYINT`	1 byte
`SMALLINT`	2 bytes
`MEDIUMINT`	3 bytes
`INT`, `INTEGER`	4 bytes
`BIGINT`	8 bytes
`FLOAT(p)`	4 bytes si 0 <= *p* <= 24, 8 bytes si 25 <= *p* <= 53
`FLOAT`	4 bytes
`DOUBLE [PRECISION]`, objeto `REAL`	8 bytes
`DECIMAL(`*M*`,D)`, `NUMERIC(M,D)`	Varía; consulte la siguiente explicación
`BIT(`*M*`)`	aproximadamente (*M*+7)/8 bytes

Requerimientos de almacenamiento para tipos de fecha y hora

Tipo de columna	Almacenamiento requerido
`DATE`	3 bytes
`DATETIME`	8 bytes
`TIMESTAMP`	4 bytes
`TIME`	3 bytes
`YEAR`	1 byte

Requerimientos de almacenamiento para tipos de cadenas de carácteres

Tipo de columna	Almacenamiento requerido
`CHAR(`*M*`)`	`M` bytes, 0 <= `M` <= 255
`VARCHAR(`*M*`)`	`L`+1 bytes, donde *L* <= `M` y 0 <= `M` <= 255
`BINARY(M)`	`M` bytes, 0 <= *M* <= 255
`VARBINARY(M)`	`L`+1 bytes, donde *L* <= `M` y 0 <= `M` <= 255
`TINYBLOB`, `TINYTEXT`	`L`+1 byte, donde *L* < 2^8

BLOB, TEXT	L+2 bytes, donde L < 2^16
MEDIUMBLOB, MEDIUMTEXT	L+3 bytes, donde L < 2^24
LONGBLOB, LONGTEXT	*L*+4 bytes, donde L < 2^32
ENUM('*value1*','value2',...)	1 o 2 bytes, dependiendo del número de valores de la enumeración (65,535 valores como máximo)
SET('*value1*','value2',...)	1, 2, 3, 4, o 8 bytes, dependiendo del número de miembros del conjunto (64 miembros como máximo)

Los tipos VARCHAR y BLOB y TEXT son de longitud variable. Para cada uno, los requerimientos de almacenamiento depende de la longitud de los valores de la (representados por *L* en la tabla precedente), en lugar que el tamaño máximo del tipo. Por ejemplo, una columna VARCHAR(10) puede tratar una cadena con una lengitud máxima de 10. El almacenamiento requerido real es la longitud de la cadena (*L*), más 1 byte para registrar la longitud de la cadena. Para la cadena 'abcd', *L* es 4 y el requerimiento de almacenamiento son 5 .

7 Funciones y operadores

7.1 Precedencias de los operadores

La precedencia de operadores se muestra en la siguiente lista, de menor a mayor precedencia. Los operadores que se muestran juntos en una línea tienen la misma precedencia.

- :=
- ||, OR, XOR
- &&, AND
- NOT
- BETWEEN, CASE, WHEN, THEN, ELSE
- =, <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
- |
- &
- <<, >>
- -, +
- *, /, DIV, %, MOD
- ^
- - (resta unaria), ~ (inversión de bit unaria)
- !
- BINARY, COLLATE

7.2 12.1.2. Parántesis

- (...)

  Use parántesis para forzar el orden de evaluación en una expresión. Por ejemplo:

```
mysql> SELECT 1+2*3;
        -> 7
mysql> SELECT (1+2)*3;
        -> 9
```

7.3 Funciones y operadores de comparación

Las operaciones de comparación dan un valor de 1 (CIERTO), 0 (FALSO), o NULL. Estas operaciones funcionan tanto para números como para cadenas de caracteres. Las cadenas de caracteres se convierten automáticamente en números y los números en cadenas cuando es necesario.

Algunas de las funciones de esta sección (tales como LEAST() y GREATEST()) retornan valores distintos a 1 (CIERTO), 0 (FALSO), o NULL. Sin embargo, el valor que retornan se basa en operaciones de comparación realizadas como describen las siguientes reglas.

MySQL compara valores usando las siguientes reglas:

- Si uno o ambos argumentos son NULL, el resultado de la comparación es NULL, excepto para el operador de comparación NULL-safe <=> .
- Si ambos argumentos en una operación de comparación son cadenas, se comparan como cadenas.
- Si ambos argumentos 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 argumentos es una columna TIMESTAMP o DATETIME y el otro argumento es una constante, la constante se convierte en timestamp antes de realizar la comparación. Esto se hace para acercarse al comportamiento de ODBC. Esto no se hace para argumentos en IN()! Para estar seguro, siempre use cadenas completas de fechas/horas al hacer comparaciones.
- En todos los otros casos, los argumentos se comparan como números con punto flotante (reales).

Por defecto, la comparación de cadenas no es sensible a mayúsculas y usa el conjunto de carácteres actual (ISO-8859-1 Latin1 por defecto, que siempre funciona bien para inglés).

Par convertir un valor a un tipo específico para una comparación, puede usar la función CAST() . Los valores de cadenas de caracteres pueden convertirse a un conjunto de caracteres distinto usando CONVERT().

7.4 Operadores lógicos

En SQL, todos los operadores lógicos se evalúan a TRUE, FALSE, o NULL (UNKNOWN). En MySQL, se implementan como 1 (TRUE), 0 (FALSE), y NULL. La mayoría de esto es común en diferentes servidores de bases de datos SQL aunque algunos servidores pueden retornar cualquier valor distinto a cero para TRUE.

- NOT, !
  NOT lógica. Se evalúa a 1 si el operando es 0, a 0 si el operando es diferente a cero, y NOT NULL retorna NULL.
- AND, &&
  AND lógica. Se evalúa a 1 si todos los operandos son distintos a cero y no NULL, a 0 si uno o más operandos son 0, de otro modo retorna NULL.
- OR, ||
  OR lógica. Cuando ambos ooperandos son no NULL, el resultado es 1 si algún operando es diferente a cero, y 0 de otro modo. Con un operando NULL el resultado es 1 si el otro operando no es cero, y NULL de otro modo. Si ambos operandos son NULL, el resultado es NULL.
- XOR
  XOR lógica. Retorna NULL si algún operando es NULL. Para operandos no NULL , evalúa a 1 si un número par de operandos es distinto a cero, sino retorna 0.

7.5 Funciones de control de flujo

- CASE value WHEN [compare-value] THEN *result* [WHEN [compare-value] THEN result ...] [ELSE result] END, CASE WHEN [*condition*] THEN result [WHEN [*condition*] THEN result ...] [ELSE result] END
  La primera versión retorna *result* donde *value=compare-value*. La segunda versión retorna el resultado para la primera condición que es cierta. Si no hay ningún resultado coincidente, el resultado tras ELSE se retorna, o NULL si no hay parte ELSE.
  mysql> SELECT CASE 1 WHEN 1 THEN 'one'
  -> WHEN 2 THEN 'two' ELSE 'more' END;
  -> 'one'
- IF(*expr1*,expr2,expr3)
  Si expr1 es TRUE (*expr1* <> 0 and expr1 <> NULL) entonces IF() retorna *expr2*; de otro modo retorna expr3. IF() retorna un valor numérico o cadena de carácteres, en función del contexto en que se usa.
  mysql> SELECT IF(1>2,2,3);
  -> 3
- IFNULL(*expr1*,expr2)

Si *expr1* no es NULL, IFNULL() retorna *expr1*, de otro modo retorna expr2. IFNULL() retorna un valor numérico o de cadena de carácteres, en función del contexto en que se usa.
mysql> SELECT IFNULL(1,0);
-> 1
mysql> SELECT IFNULL(NULL,10);
-> 10

- NULLIF(*expr1*,*expr2*)
Retorna NULL si expr1 = expr2 es cierto, de otro modo retorna *expr1*. Es lo mismo que CASE WHEN expr1 = expr2 THEN NULL ELSE *expr1* END.
mysql> SELECT NULLIF(1,1);
-> NULL
mysql> SELECT NULLIF(1,2);
-> 1

7.6 Funciones de comparación de cadenas de caracteres

Las funciones de cadenas de carácteres retornan NULL si la longitud del resultado es mayor que el valor de la variable de sistema max_allowed_packet .

Para funciones que operan en posiciones de cadenas de carácteres, la primera posición es la 1.

- ASCII(str)
Retorna el valor numérico del carácter más a la izquierda de la cadena de carácteres str. Retorna 0 si *str* es la cadena vacía. Retorna NULL si str es NULL. ASCII() funciona para carácteres con valores numéricos de 0 a 255.
mysql> SELECT ASCII('2');
-> 50
- CONCAT(*str1*,*str2*,...)
Retorna la cadena resultado de concatenar los argumentos. Retorna NULL si algúna argumento es NULL. Puede tener uno o más argumentos. Si todos los argumentos son cadenas no binarias, el resultado es una cadena no binaria. Si los argumentos incluyen cualquier cadena binaria, el resultado es una cadena binaria. Un argumento numérico se convierte a su forma de cadena binaria equivalente; si quiere evitarlo puede usar conversión de tipos explícita, como en este ejemplo: SELECT CONCAT(CAST(int_col AS CHAR), char_col)
mysql> SELECT CONCAT('My', 'S', 'QL');
-> 'MySQL'
mysql> SELECT CONCAT('My', NULL, 'QL');
-> NULL
mysql> SELECT CONCAT(14.3);
-> '14.3'
- CONCAT_WS(separator,str1,str2,...)
CONCAT_WS() significa CONCAT With Separator (CONCAT con separador) y es una forma especial de CONCAT(). El primer argumento es el separador para el resto de argumentos. El separador se añade entre las cadenas a concatenar. El separador puede ser una cadena como el resto de argumentos. Si el separador es NULL, el resultado es NULL. La función evita valores NULL tras el argumento separador.
mysql> SELECT CONCAT_WS(',','First name','Second name','Last Name');
-> 'First name,Second name,Last Name'
mysql> SELECT CONCAT_WS(',','First name',NULL,'Last Name');
-> 'First name,Last Name'
En MySQL 5.0, CONCAT_WS() no evita cadenas vacías. (Sin embargo, evita NULLs.)
- INSTR(str,*substr*)

Retorna la posición de la primera ocurrencia de la subcadena *substr* en la cadena *str*. Es lo mismo que la forma de dos argumentos de `LOCATE()`, excepto que el orden de los argumentos es inverso.

```
mysql> SELECT INSTR('foobarbar', 'bar');
    -> 4
mysql> SELECT INSTR('xbar', 'foobar');
    -> 0
```

Esta función puede trabajar con múltiples bytes. En MySQL 5.0, sólo es sensible a mayúsculas si uno de los argumentos es una cadena binaria.

- `LENGTH(str)`
  Retorna la longitud de la cadena `str`, medida en bytes.Un carácter multi-byte cuenta como múltiples bytes. Esto significa que para cadenas que contengan cinco carácteres de dos bytes, `LENGTH()` retorna 10, mientras que `CHAR_LENGTH()` retorna 5.

```
mysql> SELECT LENGTH('text');
    -> 4
```

- `REPLACE(str,from_str,to_str)`
  Retorna la cadena *str* con todas las ocurrencias de la cadena `from_str` reemplazadas con la cadena `to_str`.

```
mysql> SELECT REPLACE('www.mysql.com', 'w', 'Ww');
    -> 'WwWwWw.mysql.com'
```

  Esta función trabaja con múltiples bytes.

- `SUBSTRING(`*str*`,`*pos*`)` , `SUBSTRING(`*str* `FROM` *pos*`)`, `SUBSTRING(str,pos,len)` , `SUBSTRING(str FROM` *pos* `FOR` *len*`)`
  Las formas sin el arguemnto `len` retornan una subcadena de la cadena *str* comenzando en la posición `pos`. Las formas con el argumento `len` retornan una subcadena de longitud `len` a partir de la cadena `str`, comenzando en la posición *pos*.Las formas que usan `FROM` son sintaxis SQL estándard. En MySQL 5.0, es posible usar valores negativos para *pos*. En este caso, el inicio de la subcadena son `pos` carácteres a partir del final de la cadena, en lugar del principio. Un valor negativo puede usarse para `pos` en cualquier de las formas de esta función.

```
mysql> SELECT SUBSTRING('Quadratically',5);
    -> 'ratically'
mysql> SELECT SUBSTRING('foobarbar' FROM 4);
    -> 'barbar'
mysql> SELECT SUBSTRING('Quadratically',5,6);
    -> 'ratica'
mysql> SELECT SUBSTRING('Sakila', -3);
    -> 'ila'
mysql> SELECT SUBSTRING('Sakila', -5, 3);
    -> 'aki'
mysql> SELECT SUBSTRING('Sakila' FROM -4 FOR 2);
    -> 'ki'
```

  Esta función trabaja con múltiples bytes.
  Tenga en cuenta que si usa un valor menor a 1 para *len*, el resultado siempre es una cadena vacía.

- `SUBSTR()` es sinónimo de `SUBSTRING()`.
- `SUBSTRING_INDEX(`*str*`,`*delim*`,count)`
  Retorna la subcadena de la cadena `str` antes de `count` ocurrencias del delimitador `delim`. Si *count* es positivo, todo a la izquierda del delimitador final (contando desde la izquierda) se retorna. Si `count` es negativo, todo a la derecha del delimitador final (contando desde la derecha) se retorna.

```
mysql> SELECT SUBSTRING_INDEX('www.mysql.com', '.', 2);
    -> 'www.mysql'
```

```
mysql> SELECT SUBSTRING_INDEX('www.mysql.com', '.', -2);
    -> 'mysql.com'
```

Esta función trabaja con múltiples bytes.

- `TRIM([{BOTH | LEADING | TRAILING} [remstr] FROM] str), TRIM(remstr FROM] str)`
  Retorna la cadena `str` con todos los prefijos y/o sufijos `remstr` eliminados. Si ninguno de los especificadores `BOTH`, `LEADING`, o se da`TRAILING`, `BOTH` se asumen. Si `remstr` es opcional y no se especifica, los espacios se eliminan.

```
mysql> SELECT TRIM('  bar   ');
    -> 'bar'
mysql> SELECT TRIM(LEADING 'x' FROM 'xxxbarxxx');
    -> 'barxxx'
mysql> SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx');
    -> 'bar'
mysql> SELECT TRIM(TRAILING 'xyz' FROM 'barxxyz');
    -> 'barx'
```

Esta función trabaja con múltiples bytes.

7.7 Funciones de comparación de cadenas de caracteres

MySQL convierte automáticamente números a cadenas según es necesario y viceversa.

```
mysql> SELECT 1+'1';
    -> 2
mysql> SELECT CONCAT(2,' test');
    -> '2 test'
```

Si quiere convertir un número a cadena explícitamente, use la función `CAST()` :

```
mysql> SELECT 38.8, CAST(38.8 AS CHAR);
    -> 38.8, '38.8'
```

Si una función de cadenas da una cadena binaria como argumento, la cadena resultante también es binaria. Un número convertido a cadena se trata como cadena binaria (esto es, es sensible a mayúsculas en comparaciones). Esto afecta sólo a comparaciones.
Normalmente, si una expresión en una comparación de cadenas es sensible a mayúsculas, la comparación se realiza con sensibilidad a mayúsculas.

- *expr* `LIKE` *pat* `[ESCAPE '`*escape-char*`']`
  Coincidencia de patrones usando comparación mediante expresiones regulares SQL. Retorna 1 (`TRUE`) o 0 (`FALSE`). Si `expr` o *pat* es `NULL`, el resultado es `NULL`.
  El patrón no puede ser una cadena literal. Por ejemplo, puede especificarse como expresión de cadena o columna.
  Con `LIKE` puede usar los siguientes dos carácteres comodín en el patrón:

Carácter	Descrición
%	Coincidencia de cualquier número de carácteres, incluso cero carácteres
_	Coincide exactemente un carácter

```
mysql> SELECT 'David!' LIKE 'David_';
    -> 1
mysql> SELECT 'David!' LIKE '%D%v%';
    -> 1
```

- `expr NOT LIKE` *pat* `[ESCAPE '`*escape-char*`']`
  Es lo mismo que `NOT (expr LIKE` *pat* `[ESCAPE '`*escape-char*`'])`.

7.8 Operadores aritméticos

Los operadores aritméticos usuales están disponibles. Tenga en cuenta que en el caso de -, +, y *, el resultado se calcula con precisión BIGINT (64-bit) si ambos argumentos son enteros. Si uno de los argumentos es un entero sin signo, y los otros argumentos son también enteros, el resultado es un entero sin signo.

- +
  Suma:
  mysql> SELECT 3+5;
  -> 8
- -
  Resta:
  mysql> SELECT 3-5;
  -> -2
- -
  Menos unario. Cambia el signo del argumento.
  mysql> SELECT - 2;
  -> -2
  Nota: Si este operador se usa con BIGINT, el valor de retorno es también BIGINT. Esto significa que debe eliminar usar - con enteros que pueden ser iguales o menores a -2^63.
- *
  Multiplicación:
  mysql> SELECT 3*5;
  -> 15
  mysql> SELECT 18014398509481984*18014398509481984.0;
  -> 324518553658426726783156020576256.0
  mysql> SELECT 18014398509481984*18014398509481984;
  -> 0
  El resultado de la última expresión es incorrecto ya que el resultado de la multiplicación entera excede el rango de 64-bit de cálculos BIGINT.
- /
  División:
  mysql> SELECT 3/5;
  -> 0.60
  División por cero produce un resultado NULL:
  mysql> SELECT 102/(1-1);
  -> NULL
  Una división se calcula con aritmética BIGINT sólo en un contexto donde el resultado se convierte a entero.
- DIV
  División entera. Similar aFLOOR() pero funciona con valores BIGINT.
  mysql> SELECT 5 DIV 2;
  -> 2

7.9 Funciones matemáticas

Todas las funciones matemáticas retornan NULL en caso de error.

- CEILING(*X*), CEIL(X)
  Retorna el entero más pequeño no menor a *X*.
  mysql> SELECT CEILING(1.23);
  -> 2
  mysql> SELECT CEIL(-1.23);
  -> -1

Estas dos funciones son sinónimos. Tenga en cuenta que el valor retornado se convierte a BIGINT.

- FLOOR(X)
  Retorna el valor entero más grande pero no mayor a X.
  mysql> SELECT FLOOR(1.23);
  -> 1
  mysql> SELECT FLOOR(-1.23);
  -> -2
  Tenga en cuenta que el valor devuelto se convierte a BIGINT.
- MOD(*N*,M) , N % M, N MOD *M*
  Operación de módulo. Retorna el resto de N dividido por M.
  mysql> SELECT MOD(234, 10);
  -> 4
  mysql> SELECT 253 % 7;
  -> 1
  mysql> SELECT MOD(29,9);
  -> 2
  mysql> SELECT 29 MOD 9;
  -> 2
  Esta función puede usar valores BIGINT.
  MOD() también funciona con valores con una parte fraccional y retorna el resto exacto tras la división:
  mysql> SELECT MOD(34.5,3);
  -> 1.5
- POW(*X*,Y) , POWER(X,*Y*)
  Retorna el valor de *X* a la potencia de *Y*.
  mysql> SELECT POW(2,2);
  -> 4
  mysql> SELECT POW(2,-2);
  -> 0.25
- RAND(), RAND(N)
  Retorna un valor aleatorio en coma flotante del rango de 0 a 1.0. Si se especifica un argumento entero N, es usa como semilla, que produce una secuencia repetible.
  mysql> SELECT RAND();
  -> 0.9233482386203
  mysql> SELECT RAND(20);
  -> 0.15888261251047
  mysql> SELECT RAND();
  -> 0.63553050033332
  mysql> SELECT RAND();
  -> 0.70100469486881
  mysql> SELECT RAND(20);
  -> 0.15888261251047
  Puede usar esta función para recibir registros de forma aleatoria como se muestra aquí:
  mysql> SELECT * FROM *tbl_name* ORDER BY RAND();
  ORDER BY RAND() combinado con LIMIT es útil para seleccionar una muestra aleatoria de una conjunto de registros:
  mysql> SELECT * FROM table1, table2 WHERE a=b AND c<d
  -> ORDER BY RAND() LIMIT 1000;
  Tenga en cuenta que RAND() en una cláusula WHERE se re-evalúa cada vez que se ejecuta el WHERE.
  RAND() no pretende ser un generador de números aleatorios perfecto, pero es una forma rápida de generar números aleatorios ad hoc portable entre plataformas para la misma versión de MySQL.

- `ROUND(X)`, `ROUND(X,D)`
  Retorna el argumento `X`, redondeado al entero más cercano. Con dos argumentos, retorna `X` redondeado a *D* decimales. `D` puede ser negativo para redondear *D* dígitos a la izquierda del punto decimal del valor `X`.
  mysql> SELECT ROUND(-1.23);
  -> -1
  mysql> SELECT ROUND(-1.58);
  -> -2
  mysql> SELECT ROUND(1.58);
  -> 2
  mysql> SELECT ROUND(1.298, 1);
  -> 1.3
  mysql> SELECT ROUND(1.298, 0);
  -> 1
  mysql> SELECT ROUND(23.298, -1);
  -> 20
  El tipo de retorno es el mismo tipo que el del primer argumento (asumiendo que sea un entero, doble o decimal). Esto significa que para un argumento entero, el resultado es un entero (sin decimales).

7.10 Funciones de fecha y hora

Esta sección describe las funciones que pueden usarse para manipular valores temporales.

Aquí hay un ejemplo que usa funciones de fecha. La siguiente consulta selecciona todos los registros con un valor `date_col` dentro de los últimos 30 días:

```
mysql> SELECT something FROM tbl_name
  -> WHERE DATE_SUB(CURDATE(),INTERVAL 30 DAY) <= date_col;
```

Tenga en cuenta que la consulta también selecciona registros con fechas futuras.

Las funciones que esperan valores de fecha usualmente aceptan valores de fecha y hora e ignoran la parte de hora. Las funciones que esperan valores de hora usualmente aceptan valores de fecha y hora e ignoran la parte de fecha.

Las funciones que retornan la fecha u hora actuales se evalúan sólo una vez por consulta al principio de la ejecución de consulta. Esto significa que las referencias múltiples a una función tales como `NOW()` en una misma consulta siempre producen el mismo resultado. Este principio también se aplica a `CURDATE()`, `CURTIME()`, `UTC_DATE()`, `UTC_TIME()`, `UTC_TIMESTAMP()`, y a cualquiera de sus sinónimos.

- `DATE(expr)`
  Extrae la parte de fecha de la expresión de fecha o fecha y hora `expr`.
  mysql> SELECT DATE('2003-12-31 01:02:03');
  -> '2003-12-31'
- `DATEDIFF(`*expr*`,`*expr2*`)`
  `DATEDIFF()` retorna el número de días entre la fecha inicial `expr` y la fecha final *expr2*. `expr` y *expr2* son expresiones de fecha o de fecha y hora. Sólo las partes de fecha de los valores se usan en los cálculos.
  mysql> SELECT DATEDIFF('1997-12-31 23:59:59','1997-12-30');
  -> 1
  mysql> SELECT DATEDIFF('1997-11-30 23:59:59','1997-12-31');

```
-> -31
```

- `DATE_ADD(date,INTERVAL` *expr* `type)`, `DATE_SUB(`*date*`,INTERVAL` *expr* `type)`
  Estas funciones realizan operaciones aritméticas de fechas. `date` es un valor `DATETIME` o `DATE` especificando la fecha de inicio. *expr* es una expresión que especifica el intervalo a añadir o borrar de la fecha de inicio. *expr* es una cadena; puede comenzar con un '`-`' para intervalos negativos. *type* es una palabra clave que indica cómo debe interpretarse la expresión.
  La palabra clave `INTERVAL` y el especificador `type` no son sensibles a mayúsculas.
  La siguiente tabla muestra cómo se relacionan los argumentos *type* y `expr` :

`type` **Value**	**Expected** `expr` **Format**
MICROSECOND	MICROSECONDS
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS
YEAR	YEARS
SECOND_MICROSECOND	'SECONDS.MICROSECONDS'
MINUTE_MICROSECOND	'MINUTES.MICROSECONDS'
MINUTE_SECOND	'MINUTES:SECONDS'
HOUR_MICROSECOND	'HOURS.MICROSECONDS'
HOUR_SECOND	'HOURS:MINUTES:SECONDS'
HOUR_MINUTE	'HOURS:MINUTES'
DAY_MICROSECOND	'DAYS.MICROSECONDS'
DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'
DAY_MINUTE	'DAYS HOURS:MINUTES'
DAY_HOUR	'DAYS HOURS'
YEAR_MONTH	'YEARS-MONTHS'

- `DATE_FORMAT(date,format)`
  Formatea el valor `date` según la cadena `format` . Los siguientes especificadores pueden usarse en la cadena `format` :

Especificador	**Descripción**
%a	Día de semana abreviado (Sun..Sat)
%b	Mes abreviado (Jan..Dec)
%c	Mes, numérico (0..12)
%D	Día del mes con sufijo inglés (0th, 1st, 2nd, 3rd, ...)
%d	Día del mes numérico (00..31)
%e	Día del mes numérico (0..31)
%f	Microsegundos (000000..999999)

%H	Hora (00..23)
%h	Hora (01..12)
%I	Hora (01..12)
%i	Minutos, numérico (00..59)
%j	Día del año (001..366)
%k	Hora (0..23)
%l	Hora (1..12)
%M	Nombre mes (January..December)
%m	Mes, numérico (00..12)
%p	AM o PM
%r	Hora, 12 horas (hh:mm:ss seguido de AM o PM)
%S	Segundos (00..59)
%s	Segundos (00..59)
%T	Hora, 24 horas (hh:mm:ss)
%U	Semana (00..53), donde domingo es el primer día de la semana
%u	Semana (00..53), donde lunes es el primer día de la semana
%V	Semana (01..53), donde domingo es el primer día de la semana; usado con %X
%v	Semana (01..53), donde lunes es el primer día de la semana; usado con %x
%W	Nombre día semana (Sunday..Saturday)
%w	Día de la semana (0=Sunday..6=Saturday)
%X	Año para la semana donde domingo es el primer día de la semana, numérico, cuatro dígitos; usado con %V
%x	Año para la semana, donde lunes es el primer día de la semana, numérico, cuatro dígitos; usado con %v
%Y	Año, numérico, cuatro dígitos
%y	Año, numérico (dos dígitos)
%%	Carácter '%' literal

Todos los otros carácteres se copian al resultado sin interpretación.
Tenga en cuenta que el carácter '%' se necesita antes de carácteres especificadores de formato.
Los rangos para los especificadores de mes y día comienzan en cero debido a que MySQL permite almacenar fechas incompletas tales como '2004-00-00'.

```
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00', '%W %M %Y');
    -> 'Saturday October 1997'
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00', '%H:%i:%s');
    -> '22:23:00'
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00',
                  '%D %y %a %d %m %b %j');
    -> '4th 97 Sat 04 10 Oct 277'
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00',
                  '%H %k %I %r %T %S %w');
    -> '22 22 10 10:23:00 PM 22:23:00 00 6'
mysql> SELECT DATE_FORMAT('1999-01-01', '%X %V');
```

-> '1998 52'

- `LAST_DAY(date)`
  Toma una fecha o fecha/hora y retorna el valor correspondiente para el último día del mes. Retorna `NULL` si el argumento es inválido.
  mysql> SELECT LAST_DAY('2003-02-05');
  -> '2003-02-28'
  mysql> SELECT LAST_DAY('2004-02-05');
  -> '2004-02-29'
  mysql> SELECT LAST_DAY('2004-01-01 01:01:01');
  -> '2004-01-31'
  mysql> SELECT LAST_DAY('2003-03-32');
  -> NULL
- `NOW()`
  Retorna la fecha y hora actual como valor en formato `'YYYY-MM-DD HH:MM:SS'` o `YYYYMMDDHHMMSS` , dependiendo de si la función se usa en contexto numérico o de cadena de carácteres.
  mysql> SELECT NOW();
  -> '1997-12-15 23:50:26'
  mysql> SELECT NOW() + 0;
  -> 19971215235026
- `STR_TO_DATE(`*str*`,format)`
  Esta es la inversa de la función `DATE_FORMAT()`. Toma la cadena *str* y la cadena de formato `format`. `STR_TO_DATE()` retorna un valor `DATETIME` si la cadena de formato contiene parte de fecha y hora, o un valor `DATE` o `TIME` si la cadena contiene sólo parte de fecha o hora. Los valores fecha, hora o fecha/hora contenidos en *str* deben ser dados en el formato indicado por *format*. Para los especificadores que pueden usarse en `format`, consulte la tabla en la descripción de la función `DATE_FORMAT()` . Todos los otros carácteres no se interpretan. Si *str* contiene un valor fecha, hora o fecha/hora ilegal, `STR_TO_DATE()` retorna `NULL`. A partir de MySQL 5.0.3, un valor ilegal también produce una advertencia.

  mysql> SELECT STR_TO_DATE('03.10.2003 09.20','%d.%m.%Y %H.%i');
  -> '2003-10-03 09:20:00'
  mysql> SELECT STR_TO_DATE('10arp', '%carp');
  -> '0000-10-00 00:00:00'
  mysql> SELECT STR_TO_DATE('2003-15-10 00:00:00','%Y-%m-%d %H:%i:%s');
  -> NULL

7.11 Funciones de encriptación

Las funciones en esta sección encriptan y desencriptan valores. Si quiere almacenar resultados de una función de encriptación que puede contaner valores arbitrarios de bytes, use una columna `BLOB` en lugar de `CHAR` o `VARCHAR` para evitar problemas potenciales con eliminación de espacios finales que pueden cambiar los valores de datos.

- `DECODE(`*crypt_str*`,pass_str)`
  Desencripta la cadena encriptada `crypt_str` usando *pass_str* como contraseña. `crypt_str` debe ser una cadena retornada de `ENCODE()`.
- `ENCODE(str,pass_str)`
  Encripta `str` usando *pass_str* como contraseña. Para desencriptar el resultado, use `DECODE()`.
  El resultado es una cadena binaria de la misma longitud que *str*. Si quiere guardarlo en una columna, use una columna de tipo `BLOB` .

7.12 Funciones de información

- `COLLATION(str)`
  Retorna la colación para el conjunto de carácteres de la cadena dada.
  mysql> SELECT COLLATION('abc');
  -> 'latin1_swedish_ci'
- `CURRENT_USER()`
  Retorna la combinación de nombre de usuario y de equipo que tiene la sesión actual.
- `DATABASE()`
  Retorna el nombre de base de datos por defecto (actual). En MySQL 5.0, la cadena tiene el conjunto de carácteres `utf8` .
- `LAST_INSERT_ID()`, `LAST_INSERT_ID(expr)`
  Retorna el último valor generado automáticamente que se insertó en una columna `AUTO_INCREMENT`.
  mysql> SELECT LAST_INSERT_ID();
  -> 195
  El último ID generado se mantiene en el servidor para cada conexión. Esto significa que el valor de la función retorna a cada cliente el valor `AUTO_INCREMENT` más reciente generado por ese cliente. Este valor no puede ser afectado por otros clientes, incluso si generan valores `AUTO_INCREMENT` ellos mismos. Este comportamiento asegura que reciba sus propios IDs sin tener en cuenta la actividad de otros clientes y sin la necesidad de bloqueos o transacciones. El valor de `LAST_INSERT_ID()` no cambia si actualiza la columna `AUTO_INCREMENT` de un registro con un valor no mágico (esto es, un valor que no es `NULL` ni `0`).
  Si inserta varios registros a la vez con un comando de inserción `LAST_INSERT_ID()` retorna el valor del primer registro insertado. La razón para esto es hacer posible reproducir fácilmente el mismo comando `INSERT` contra otro servidor.

- `ROW_COUNT()`
  `ROW_COUNT()` retorna el número de registros actualizados, insertados o borrados por el comando precedente. Esto es lo mismo que el número de registros que muestra el cliente **mysql** y el valor de la función de la API C `mysql_affected_rows()` .
  mysql> INSERT INTO t VALUES(1),(2),(3);
  Query OK, 3 rows affected (0.00 sec)
  Records: 3 Duplicates: 0 Warnings: 0

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

- `VERSION()`
  Retorna una cadena que indica la versión del servidor MySQL. La cadena usa el conjunto de carácteres `utf8` .

8 Procedimientos y funciones

8.1 Uso

Los procedimientos son códigos que se ejecutan directamente en el servidor.

Un procedimiento se invoca usando un comando CALL , y sólo puede pasar valores usando variables de salida. Una función puede llamarse desde desde dentro de un comando como cualquier otra función (esto es, invocando el nombre de la función), y puede retornar un valor escalar. Las rutinas almacenadas pueden llamar otras rutinas almacenadas.

8.2 CREATE PROCEDURE y CREATE FUNCTION

CREATE PROCEDURE *sp_name* ([parameter[,...]])
  [characteristic ...] *routine_body*

CREATE FUNCTION sp_name ([parameter[,...]])
  RETURNS *type*
  [*characteristic* ...] routine_body

parameter:
  [IN | OUT | INOUT] *param_name type*

type:
  Any valid MySQL data type

characteristic:
  LANGUAGE SQL
 | [NOT] DETERMINISTIC
 | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
 | SQL SECURITY { DEFINER | INVOKER }
 | COMMENT 'string'

routine_body:
  procedimientos almacenados o comandos SQL válidos

Estos comandos crean una rutina almacenada.

Por defecto, la rutina se asocia con la base de datos actual.

Para asociar la rutina explícitamente con una base de datos, especifique el nombre como db_name.sp_name al crearlo.

Si el nombre de rutina es el mismo que el nombre de una función de SQL, necesita usar un espacio entre el nombre y el siguiente paréntesis al definir la rutina, o hay un error de sintaxis. Esto también es cierto cuando invoca la rutina posteriormente.

La cláusula RETURNS puede especificarse sólo con FUNCTION, donde es obligatorio. Se usa para indicar el tipo de retorno de la función, y el cuerpo de la función debe contener un comando RETURN value.

La lista de parámetros entre paréntesis debe estar siempre presente. Si no hay parámetros, se debe usar una lista de parámetros vacía `()` .

Cada parámetro es un parámetro `IN` por defecto. Para especificar otro tipo de parámetro, use la palabra clave `OUT` o `INOUT` antes del nombre del parámetro. Especificando `IN`, `OUT`, o `INOUT` sólo es valido para una `PROCEDURE`.

Un marco para procedimientos almacenados externos se introducirá en el futuro. Esto permitira escribir procedimientos almacenados en lenguajes distintos a SQL. Uno de los primeros lenguajes a soportar será PHP ya que el motor central de PHP es pequeño, con flujos seguros y puede empotrarse fácilmente. Como el marco es público, se espera soportar muchos otros lenguajes.

Un procedimiento o función se considera "determinista" si siempre produce el mismo resultado para los mismos parámetros de entrada, y "no determinista" en cualquier otro caso. Si no se da ni `DETERMINISTIC` ni `NOT DETERMINISTIC` por defecto es `NOT DETERMINISTIC`.

Varias características proporcionan información sobre la naturaleza de los datos usados por la rutina. `CONTAINS SQL` indica que la rutina no contiene comandos que leen o escriben datos. `NO SQL` indica que la rutina no contiene comandos SQL . `READS SQL DATA` indica que la rutina contiene comandos que leen datos, pero no comandos que escriben datos. `MODIFIES SQL DATA` indica que la rutina contiene comandos que pueden escribir datos. `CONTAINS SQL` es el valor por defecto si no se dan explícitamente ninguna de estas características.

La característica `SQL SECURITY` puede usarse para especificar si la rutina debe ser ejecutada usando los permisos del usuario que crea la rutina o el usuario que la invoca. El valor por defecto es `DEFINER`. Esta característica es nueva en SQL:2003. El creador o el invocador deben tener permisos para acceder a la base de datos con la que la rutina está asociada.

MySQL almacena la variable de sistema `sql_mode` que está en efecto cuando se crea la rutina, y siempre ejecuta la rutina con esta inicialización.

La cláusula `COMMENT` es una extensión de MySQL, y puede usarse para describir el procedimiento almacenado. Esta información se muestra con los comandos `SHOW CREATE PROCEDURE` y `SHOW CREATE FUNCTION` .

MySQL permite a las rutinas que contengan comandos DDL (tales como `CREATE` y `DROP`) y comandos de transacción SQL (como `COMMIT`). Esto no lo requiere el estándar, y por lo tanto, es específico de la implementación.

Los procedimientos almacenados no pueden usar `LOAD DATA INFILE`.

Nota: Actualmente, los procedimientos almacenados creados con `CREATE FUNCTION` no pueden tener referencias a tablas. (Esto puede incluir algunos comandos `SET` que pueden contener referencias a tablas, por ejemplo `SET a:= (SELECT MAX(id) FROM t)`, y por otra parte no pueden contener comandos `SELECT` , por ejemplo `SELECT 'Hello world!' INTO var1`.) Esta limitación se elminará en breve.

Los comandos que retornan un conjunto de resultados no pueden usarse desde una función almacenada. Esto incluye comandos `SELECT` que no usan `INTO` para tratar valores de columnas en variables, comandos `SHOW` y otros comandos como `EXPLAIN`. Para comandos que pueden determinarse al definir la función para que retornen un conjunto de resultados, aparece un mensaje de error `Not allowed to return a result set from a function` (`ER_SP_NO_RETSET_IN_FUNC`). Para comandos que puede determinarse sólo en tiempo de ejecución si retornan un conjunto de resultados,

aparece el error `PROCEDURE %s can't return a result set in the given context` (`ER_SP_BADSELECT`).

El siguiente es un ejemplo de un procedimiento almacenado que use un parámetro `OUT` . El ejemplo usa el cliente **mysql** y el comando `delimiter` para cambiar el delimitador del comando de `;` a `//` mientras se define el procedimiento . Esto permite pasar el delimitador `;` usado en el cuerpo del procedimiento a través del servidor en lugar de ser interpretado por el mismo **mysql**.

```
mysql> delimiter //

mysql> CREATE PROCEDURE simpleproc (OUT param1 INT)
   -> BEGIN
   ->   SELECT COUNT(*) INTO param1 FROM t;
   -> END
   -> //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;

mysql> CALL simpleproc(@a);
Query OK, 0 rows affected (0.00 sec)

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

Al usar el comando `delimiter`, debe evitar el uso de la antibarra ('\') ya que es el carácter de escape de MySQL.

El siguiente es un ejemplo de función que toma un parámetro, realiza una operación con una función SQL, y retorna el resultado:

```
mysql> delimiter //

mysql> CREATE FUNCTION hello (s CHAR(20)) RETURNS CHAR(50)
   -> RETURN CONCAT('Hello, ',s,'!');
   -> //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;

mysql> SELECT hello('world');
+----------------+
| hello('world') |
+----------------+
| Hello, world!  |
+----------------+
1 row in set (0.00 sec)
```

Si el comando `RETURN` en un procedimiento almacenado retorna un valor con un tipo distinto al especificado en la cláusula `RETURNS` de la función, el valor de retorno se coherciona al tipo apropiado. Por ejemplo, si una función retorna un valor `ENUM` o `SET`, pero el comando `RETURN` retorna un entero, el valor retornado por la función es la cadena para el mienbro de `ENUM` correspondiente de un comjunto de miembros `SET` .

8.3 La sentencia CALL

CALL sp_name([parameter[,...]])

El comando CALL invoca un procedimiento definido préviamente con CREATE PROCEDURE.
CALL puede pasar valores al llamador usando parámetros declarados como OUT o INOUT . También "retorna" el número de registros afectados, que con un programa cliente puede obtenerse a nivel SQL llamando la función ROW_COUNT() y desde C llamando la función de la API C mysql_affected_rows() .

8.4 Sentencia compuesta BEGIN ... END

[*begin_label*:] BEGIN
[*statement_list*]
END [end_label]

Los procedimientos almacenados pueden contener varios comandos, usnado un comando compuesto BEGIN ... END .

Un comando compuesto puede etiquetarse. *end_label* no puede darse a no ser que también esté presente *begin_label* , y si ambos lo están, deben ser el mismo.
Tenga en cuenta que la cláusula opcional [NOT] ATOMIC no está soportada. Esto significa que no hay un punto transaccional al inicio del bloque de instrucciones y la cláusula BEGIN usada en este contexto no tiene efecto en la transacción actual.

Usar múltiples comandos requiere que el cliente sea capaz de enviar cadenas de consultas con el delimitador de comando ;. Esto se trata en el cliente de línea de comandos **mysql** con el comando delimiter. Cambiar el delimitador de final de consulta ; end-of-query (por ejemplo, a //) permite usar ; en el cuerpo de la rutina.

8.5 Sentencia DECLARE

El comando DECLARE se usa para definir varios iconos locales de una rutina: las variables locales, condiciones y handlers y cursores. Los comandos SIGNAL y RESIGNAL no se soportan en la actualidad.
DECLARE puede usarse sólo dentro de comandos compuestos BEGIN ... END y deben ser su inicio, antes de cualquier otro comando.

Los cursores deben declararse antes de declarar los handlers, y las variables y condiciones deben declararse antes de declarar los cursores o handlers.

8.6 Variables en procedimientos almacenados

8.6.1 Declarar variables locales con DECLARE

DECLARE var_name[,...] *type* [DEFAULT *value*]
Este comando se usa para declarar variables locales. Para proporcionar un valor por defecto para la variable, incluya una cláusula DEFAULT . El valor puede especificarse como expresión, no necesita ser una constante. Si la cláusula DEFAULT no está presente, el valor inicial es NULL.

La visibilidad de una variable local es dentro del bloque `BEGIN ... END` donde está declarado. Puede usarse en bloques anidados excepto aquéllos que declaren una variable con el mismo nombre.

8.6.2 Sentencia SET para variables

SET *var_name* = *expr* [, `var_name` = *expr*] ...
El comando `SET` en procedimientos almacenados es una versión extendida del comando general `SET`.

Las variables referenciadas pueden ser las declaradas dentro de una rutina, o variables de servidor globales.

El comando `SET` en procedimientos almacenados se implementa como parte de la sintaxis `SET` pre-existente. Esto permite una sintaxis extendida de `SET a=x, b=y, ...` donde distintos tipos de variables (variables declaradas local y globalmente y variables de sesión del servidor) pueden mezclarse. Esto permite combinaciones de variables locales y algunas opciones que tienen sentido sólo para variables de sistema; en tal caso, las opciones se reconocen pero se ignoran.

8.6.3 La sentencia SELECT ... INTO

SELECT `col_name`[,...] INTO *var_name*[,...] `table_expr`

Esta sintaxis `SELECT` almacena columnas seleccionadas directamente en variables. Por lo tanto, sólo un registro puede retornarse.

SELECT id,data INTO x,y FROM test.t1 LIMIT 1;

8.6.4 Conditions and Handlers

Ciertas condiciones pueden requerir un tratamiento específico. Estas condiciones pueden estar relacionadas con errores, así como control de flujo general dentro de una rutina.

8.6.5 Condiciones DECLARE

DECLARE `condition_name` CONDITION FOR *condition_value*

condition_value:
  SQLSTATE [VALUE] *sqlstate_value*
 | `mysql_error_code`

Este comando especifica condiciones que necesitan tratamiento específico. Asocia un nombre con una condición de error específica. El nombre puede usarse subsecuentemente en un comando `DECLARE HANDLER` .

Además de valores SQLSTATE , los códigos de error MySQL se soportan.

8.6.6 DECLARE handlers

DECLARE *handler_type* HANDLER FOR *condition_value*[,...] *sp_statement*

handler_type:
  CONTINUE
 | EXIT
 | UNDO

condition_value:
  SQLSTATE [VALUE] sqlstate_value
 | condition_name
 | SQLWARNING
 | NOT FOUND
 | SQLEXCEPTION
 | *mysql_error_code*

Este comando especifica handlers que pueden tratar una o varias condiciones. Si una de estas condiciones ocurren, el comando especificado se ejecuta.

Para un handler CONTINUE , continúa la rutina actual tras la ejecución del comando del handler. Para un handler EXIT , termina la ejecución del comando compuesto BEGIN...END actual. El handler de tipo UNDO todavía no se soporta.

- SQLWARNING es una abreviación para todos los códigos SQLSTATE que comienzan con 01.
- NOT FOUND es una abreviación para todos los códigos SQLSTATE que comienzan con 02.
- SQLEXCEPTION es una abreviación para todos los códigos SQLSTATE no tratados por SQLWARNING o NOT FOUND.

Además de los valores SQLSTATE , los códigos de error MySQL se soportan.
Por ejemplo:

```
mysql> CREATE TABLE test.t (s1 int,primary key (s1));
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter //

mysql> CREATE PROCEDURE handlerdemo ()
   -> BEGIN
   ->   DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;
   ->   SET @x = 1;
   ->   INSERT INTO test.t VALUES (1);
   ->   SET @x = 2;
   ->   INSERT INTO test.t VALUES (1);
   ->   SET @x = 3;
   -> END;
   -> //
Query OK, 0 rows affected (0.00 sec)

mysql> CALL handlerdemo()//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x//
   +------+
   | @x   |
   +------+
   | 3    |
   +------+
   1 row in set (0.00 sec)
```

Tenga en cuenta que @x es 3, lo que muestra que MySQL se ha ejecutado al final del procedimiento. Si la línea DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1; no está presente, MySQL habría tomado la ruta por defecto (EXIT) tras el segundo INSERT fallido debido a la restricción PRIMARY KEY , y SELECT @x habría retornado 2.

8.7 Cursores

Se soportan cursores simples dentro de procedimientos y funciones almacenadas. La sintaxis es la de SQL empotrado. Los cursores no son sensibles, son de sólo lectura, y no permiten scrolling. No sensible significa que el servidor puede o no hacer una copia de su tabla de resultados.

Los cursores deben declararse antes de declarar los handlers, y las variables y condiciones deben declararse antes de declarar cursores o handlers.
Por ejemplo:

```
CREATE PROCEDURE curdemo()
BEGIN
  DECLARE done INT DEFAULT 0;
  DECLARE a CHAR(16);
  DECLARE b,c INT;
  DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
  DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;

  OPEN cur1;
  OPEN cur2;

  REPEAT
    FETCH cur1 INTO a, b;
    FETCH cur2 INTO c;
    IF NOT done THEN
      IF b < c THEN
        INSERT INTO test.t3 VALUES (a,b);
      ELSE
        INSERT INTO test.t3 VALUES (a,c);
      END IF;
    END IF;
  UNTIL done END REPEAT;

  CLOSE cur1;
  CLOSE cur2;
END
```

8.7.1 Declarar cursores

DECLARE `cursor_name` CURSOR FOR `select_statement`

Este comando declara un cursor. Pueden definirse varios cursores en una rutina, pero cada cursor en un bloque debe tener un nombre único.

El comando `SELECT` no puede tener una cláusula `INTO` .

8.7.2 Sentencia `OPEN` del cursor

OPEN `cursor_name`

Este comando abre un cursor declarado préviamente.

8.7.3 Sentencia de cursor `FETCH`

FETCH *cursor_name* INTO `var_name` [, `var_name`] ...

Este comando trata el siguiente registro (si existe) usando el cursor abierto especificado, y avanza el puntero del cursro.

8.7.4 Sentencia de cursor `CLOSE`

CLOSE `cursor_name`

Este comando cierra un cursor abierto préviamente.
Si no se cierra explícitamente, un cursor se cierra al final del comando compuesto en que se declara.

8.8 Constructores de control de flujo

Los constructores `IF`, `CASE`, `LOOP`, `WHILE`, `ITERATE`, y `LEAVE` están completamente implementados. Estos constructores pueden contener un comando simple, o un bloque de comandos usando el comando compuesto `BEGIN ... END`. Los constructores pueden estar anidados.
Los bucles `FOR` no están soportados.

8.8.1 Sentencia `IF`

IF *search_condition* THEN `statement_list`
  [ELSEIF `search_condition` THEN *statement_list*] ...
  [ELSE *statement_list*]
END IF

`IF` implementa un constructor condicional básico. Si *search_condition* se evalúa a cierto, el comando SQL correspondiente listado se ejectua. Si no coincide ninguna *search_condition* se ejecuta el comando listado en la cláusula `ELSE`. `statement_list` puede consistir en varios comandos.
Tenga en cuenta que también hay una *función* `IF()`, que difiere del *commando* `IF` descrito aquí.

8.8.2 La sentencia `CASE`

CASE *case_value*
  WHEN `when_value` THEN *statement_list*
  [WHEN *when_value* THEN `statement_list`] ...
  [ELSE *statement_list*]
END CASE
O:
CASE
  WHEN `search_condition` THEN `statement_list`
  [WHEN *search_condition* THEN `statement_list`] ...
  [ELSE `statement_list`]
END CASE

El comando `CASE` para procedimientos almacenados implementa un constructor condicional complejo. Si una `search_condition` se evalúa a cierto, el comando SQL correspondiente se ejecuta. Si no coincide ninguna condición de búsqueda, el comando en la cláusula `ELSE` se ejecuta.

Nota: La sitaxis de un *comando* `CASE` mostrado aquí para uso dentro de procedimientos almacenados difiere ligeramente de la *expresión* `CASE` SQL descrita en la Sección "Funciones de control de flujo". El comando `CASE` no puede tener una cláusula `ELSE NULL` y termina con `END CASE` en lugar de `END`.

8.8.3 Sentencia LOOP

[begin_label:] LOOP
  statement_list
END LOOP [end_label]

LOOP implementa un constructor de bucle simple que permite ejecución repetida de comandos particulares. El comando dentro del bucle se repite hasta que acaba el bucle, usualmente con un comando LEAVE .

Un comando LOOP puede etiquetarse. end_label no puede darse hasta que esté presente *begin_label* , y si ambos lo están, deben ser el mismo.

8.8.4 Sentencia LEAVE

LEAVE *label*

Este comando se usa para abandonar cualquier control de flujo etiquetado. Puede usarse con BEGIN ... END o bucles.

8.8.5 La setencia ITERATE

ITERATE *label*

ITERATE sólo puede aparecer en comandos LOOP, REPEAT, y WHILE . ITERATE significa "vuelve a hacer el bucle."

Por ejemplo:
```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
 label1: LOOP
   SET p1 = p1 + 1;
   IF p1 < 10 THEN ITERATE label1; END IF;
   LEAVE label1;
 END LOOP label1;
 SET @x = p1;
END
```

8.8.6 Sentencia REPEAT

[*begin_label*:] REPEAT
  statement_list
UNTIL search_condition
END REPEAT [end_label]

El comando/s dentro de un comando REPEAT se repite hasta que la condición search_condition es cierta.

Un comando REPEAT puede etiquetarse. end_label no puede darse a no ser que *begin_label* esté presente, y si lo están, deben ser el mismo.

Por ejemplo:
```
mysql> delimiter //
```

```
mysql> CREATE PROCEDURE dorepeat(p1 INT)
    -> BEGIN
    ->   SET @x = 0;
    ->   REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
    -> END
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> CALL dorepeat(1000)//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x//
+------+
| @x   |
+------+
| 1001 |
+------+
1 row in set (0.00 sec)
```

8.8.7 Sentencia `WHILE`

[`begin_label`:] WHILE `search_condition` DO
  statement_list
END WHILE [*end_label*]

El comado/s dentro de un comando `WHILE` se repite mientras la condición `search_condition` es cierta.

Un comando `WHILE` puede etiquetarse. `end_label` no puede darse a no ser que *begin_label* también esté presente, y si lo están, deben ser el mismo.

Por ejemplo:

```
CREATE PROCEDURE dowhile()
BEGIN
 DECLARE v1 INT DEFAULT 5;

 WHILE v1 > 0 DO
  ...
  SET v1 = v1 - 1;
 END WHILE;
END
```


9 Disparadores

Un disparador es un objeto con nombre dentro de una base de datos el cual se asocia con una tabla y se activa cuando ocurre en ésta un evento en particular. Por ejemplo, las siguientes sentencias crean una tabla y un disparador para sentencias INSERT dentro de la tabla. El disparador suma los valores insertados en una de las columnas de la tabla:

```
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account
   -> FOR EACH ROW SET @sum = @sum + NEW.amount;
```

Este capítulo describe la sintaxis para crear y eliminar disparadores, y muestra algunos ejemplos de cómo utilizarlos.

9.1 Sintaxis de CREATE TRIGGER

CREATE TRIGGER `nombre_disp` *momento_disp* `evento_disp`
ON *nombre_tabla* FOR EACH ROW *sentencia_disp*

Un disparador es un objeto con nombre en una base de datos que se asocia con una tabla, y se activa cuando ocurre un evento en particular para esa tabla.

El disparador queda asociado a la tabla `nombre_tabla`. Esta debe ser una tabla permanente, no puede ser una tabla `TEMPORARY` ni una vista.

`momento_disp` es el momento en que el disparador entra en acción. Puede ser `BEFORE` (antes) o `AFTER` (despues), para indicar que el disparador se ejecute antes o después que la sentencia que lo activa.

`evento_disp` indica la clase de sentencia que activa al disparador. Puede ser `INSERT`, `UPDATE`, o `DELETE`. Por ejemplo, un disparador `BEFORE` para sentencias `INSERT` podría utilizarse para validar los valores a insertar.

No puede haber dos disparadores en una misma tabla que correspondan al mismo momento y sentencia. Por ejemplo, no se pueden tener dos disparadores `BEFORE UPDATE`. Pero sí es posible tener los disparadores `BEFORE UPDATE` y `BEFORE INSERT` o `BEFORE UPDATE` y `AFTER UPDATE`.

sentencia_disp es la sentencia que se ejecuta cuando se activa el disparador. Si se desean ejecutar múltiples sentencias, deben colocarse entre `BEGIN ... END`, el constructor de sentencias compuestas.

Esto además posibilita emplear las mismas sentencias permitidas en rutinas almacenadas.

Las columnas de la tabla asociada con el disparador pueden referenciarse empleando los alias `OLD` y `NEW`. `OLD.nombre_col` hace referencia a una columna de una fila existente, antes de ser actualizada o borrada. `NEW.nombre_col` hace referencia a una columna en una nueva fila a punto de ser insertada, o en una fila existente luego de que fue actualizada.

El uso de `SET NEW.`*nombre_col* `= valor` necesita que se tenga el privilegio `UPDATE` sobre la columna. El uso de `SET nombre_var = NEW.nombre_col` necesita el privilegio `SELECT` sobre la columna.

Nota: Actualmente, los disparadores no son activados por acciones llevadas a cabo en cascada por las restricciones de claves extranjeras. Esta limitación se subsanará tan pronto como sea posible.

9.2 Sintaxis de DROP TRIGGER

DROP TRIGGER [*nombre_esquema*.]nombre_disp

Elimina un disparador. El nombre de esquema es opcional. Si el esquema se omite, el disparador se elimina en el esquema actual.

9.3 Utilización de disparadores

El soporte para disparadores se incluyó a partir de MySQL 5.0.2. Actualmente, el soporte para disparadores es básico, por lo tanto hay ciertas limitaciones en lo que puede hacerse con ellos. Esta sección trata sobre el uso de los disparadores y las limitaciones vigentes.

Un disparador es un objeto de base de datos con nombre que se asocia a una tabla, y se activa cuando ocurre un evento en particular para la tabla. Algunos usos para los disparadores es verificar valores a ser insertados o llevar a cabo cálculos sobre valores involucrados en una actualización.

Un disparador se asocia con una tabla y se define para que se active al ocurrir una sentencia INSERT, DELETE, o UPDATE sobre dicha tabla. Puede también establecerse que se active antes o despues de la sentencia en cuestión. Por ejemplo, se puede tener un disparador que se active antes de que un registro sea borrado, o después de que sea actualizado.

Este es un ejemplo sencillo que asocia un disparador con una tabla para cuando reciba sentencias INSERT. Actúa como un acumulador que suma los valores insertados en una de las columnas de la tabla.

La siguiente sentencia crea la tabla y un disparador asociado a ella:

```
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account
   -> FOR EACH ROW SET @sum = @sum + NEW.amount;
```

La sentencia CREATE TRIGGER crea un disparador llamado ins_sum que se asocia con la tabla account. También se incluyen cláusulas que especifican el momento de activación, el evento activador, y qué hacer luego de la activación:

- La palabra clave BEFORE indica el momento de acción del disparador. En este caso, el disparador debería activarse antes de que cada registro se inserte en la tabla. La otra palabra clave posible aqui es AFTER.
- La plabra clave INSERT indica el evento que activará al disparador. En el ejemplo, la sentencia INSERT causará la activación. También pueden crearse disparadores para sentencias DELETE y UPDATE.
- Las sentencia siguiente, FOR EACH ROW, define lo que se ejecutará cada vez que el disparador se active, lo cual ocurre una vez por cada fila afectada por la sentencia activadora. En el ejemplo, la sentencia activada es un sencillo SET que acumula los valores insertados en la columna amount. La sentencia se refiere a la columna como NEW.amount, lo que significa "el valor de la columna amount que será insertado en el nuevo registro."

Para utilizar el disparador, se debe establecer el valor de la variable acumulador a cero, ejecutar una sentencia INSERT, y ver qué valor presenta luego la variable.

```
mysql> SET @sum = 0;
mysql> INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);
mysql> SELECT @sum AS 'Total amount inserted';
+----------------------+
| Total amount inserted |
+----------------------+
| 1852.48              |
+----------------------+
```

En este caso, el valor de `@sum` luego de haber ejecutado la sentencia `INSERT` es `14.98 + 1937.50 - 100`, o `1852.48`.

Para eliminar el disparador, se emplea una sentencia `DROP TRIGGER`. El nombre del disparador debe incluir el nombre de la tabla:

```
mysql> DROP TRIGGER account.ins_sum;
```

Debido a que un disparador está asociado con una tabla en particular, no se pueden tener múltiples disparadores con el mismo nombre dentro de una tabla. También se debería tener en cuenta que el espacio de nombres de los disparadores puede cambiar en el futuro de un nivel de tabla a un nivel de base de datos, es decir, los nombres de disparadores ya no sólo deberían ser únicos para cada tabla sino para toda la base de datos. Para una mejor compatibilidad con desarrollos futuros, se debe intentar emplear nombres de disparadores que no se repitan dentro de la base de datos.

Adicionalmente al requisito de nombres únicos de disparador en cada tabla, hay otras limitaciones en los tipos de disparadores que pueden crearse. En particular, no se pueden tener dos disparadores para una misma tabla que sean activados en el mismo momento y por el mismo evento. Por ejemplo, no se pueden definir dos `BEFORE INSERT` o dos `AFTER UPDATE` en una misma tabla. Es improbable que esta sea una gran limitación, porque es posible definir un disparador que ejecute múltiples sentencias empleando el constructor de sentencias compuestas `BEGIN ... END` luego de `FOR EACH ROW`. (Más adelante en esta sección puede verse un ejemplo).

También hay limitaciones sobre lo que puede aparecer dentro de la sentencia que el disparador ejecutará al activarse:

- El disparador no puede referirse a tablas directamente por su nombre, incluyendo la misma tabla a la que está asociado. Sin embargo, se pueden emplear las palabras clave `OLD` y `NEW`. `OLD` se refiere a un registro existente que va a borrarse o que va a actualizarse antes de que esto ocurra. `NEW` se refiere a un registro nuevo que se insertará o a un registro modificado luego de que ocurre la modificación.
- El disparador no puede invocar procedimientos almacenados utilizando la sentencia `CALL`. (Esto significa, por ejemplo, que no se puede utilizar un procedimiento almacenado para eludir la prohibición de referirse a tablas por su nombre).
- El disparador no puede utilizar sentencias que inicien o finalicen una transacción, tal como `START TRANSACTION`, `COMMIT`, o `ROLLBACK`.

Las palabras clave `OLD` y `NEW` permiten acceder a columnas en los registros afectados por un disparador. (`OLD` y `NEW` no son sensibles a mayúsculas). En un disparador para `INSERT`, solamente puede utilizarse `NEW.nom_col`; ya que no hay una versión anterior del registro. En un disparador para `DELETE` sólo puede emplearse `OLD.nom_col`, porque no hay un nuevo registro. En un disparador para `UPDATE` se puede emplear `OLD.nom_col` para referirse a las columnas de un registro antes de que sea actualizado, y `NEW.`*`nom_col`* para referirse a las columnas del registro luego de actualizarlo.

Una columna precedida por OLD es de sólo lectura. Es posible hacer referencia a ella pero no modificarla. Una columna precedida por NEW puede ser referenciada si se tiene el privilegio SELECT sobre ella. En un disparador BEFORE, también es posible cambiar su valor con SET NEW.nombre_col = *valor* si se tiene el privilegio de UPDATE sobre ella. Esto significa que un disparador puede usarse para modificar los valores antes que se inserten en un nuevo registro o se empleen para actualizar uno existente.

En un disparador BEFORE, el valor de NEW para una columna AUTO_INCREMENT es 0, no el número secuencial que se generará en forma automática cuando el registro sea realmente insertado. OLD y NEW son extensiones de MySQL para los disparadores.

Empleando el constructor BEGIN ... END, se puede definir un disparador que ejecute sentencias múltiples. Dentro del bloque BEGIN, también pueden utilizarse otras sintaxis permitidas en rutinas almacenadas, tales como condicionales y bucles. Como sucede con las rutinas almacenadas, cuando se crea un disparador que ejecuta sentencias múltiples, se hace necesario redefinir el delimitador de sentencias si se ingresará el disparador a través del programa **mysql**, de forma que se pueda utilizar el caracter ';' dentro de la definición del disparador. El siguiente ejemplo ilustra estos aspectos. En él se crea un disparador para UPDATE, que verifica los valores utilizados para actualizar cada columna, y modifica el valor para que se encuentre en un rango de 0 a 100. Esto debe hacerse en un disparador BEFORE porque los valores deben verificarse antes de emplearse para actualizar el registro:

```
mysql> delimiter //
mysql> CREATE TRIGGER upd_check BEFORE UPDATE ON account
   -> FOR EACH ROW
   -> BEGIN
   ->     IF NEW.amount < 0 THEN
   ->         SET NEW.amount = 0;
   ->     ELSEIF NEW.amount > 100 THEN
   ->         SET NEW.amount = 100;
   ->     END IF;
   -> END;//
mysql> delimiter ;
```

Podría parecer más fácil definir una rutina almacenada e invocarla desde el disparador utilizando una simple sentencia CALL. Esto sería ventajoso también si se deseara invocar la misma rutina desde distintos disparadores. Sin embargo, una limitación de los disparadores es que no pueden utilizar CALL. Se debe escribir la sentencia compuesta en cada CREATE TRIGGER donde se la desee emplear.

MySQL gestiona los errores ocurridos durante la ejecución de disparadores de esta manera:

- Si lo que falla es un disparador BEFORE, no se ejecuta la operación en el correspondiente registro.
- Un disparador AFTER se ejecuta solamente si el disparador BEFORE (de existir) y la operación se ejecutaron exitosamente.
- Un error durante la ejecución de un disparador BEFORE o AFTER deriva en la falla de toda la sentencia que provocó la invocación del disparador.
- En tablas transaccionales, la falla de un disparador (y por lo tanto de toda la sentencia) debería causar la cancelación (rollback) de todos los cambios realizados por esa sentencia. En tablas no transaccionales, cualquier cambio real

10 Vistas

Las vistas (incluyendo vistas actualizables) fueron introducidas en la versión 5.0 del servidor de base de datos MySQL

En este capítulo se tratan los siguientes temas:

- Creación o modificación de vistas con `CREATE VIEW` o `ALTER VIEW`
- Eliminación de vistas con `DROP VIEW`
- Obtención de información de definición de una vista (metadatos) con `SHOW CREATE VIEW`

10.1 Sintaxis de `ALTER VIEW`

ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
VIEW *nombre_vista* [(`columnas`)]
AS *sentencia_select*
[WITH [CASCADED | LOCAL] CHECK OPTION]

Esta sentencia modifica la definición de una vista existente. La sintaxis es semejante a la empleada en `CREATE VIEW`. Se requiere que posea los permisos `CREATE VIEW` y `DELETE` para la vista, y algún privilegio en cada columna seleccionada por la sentencia `SELECT`.

10.2 Sintaxis de `CREATE VIEW`

CREATE [OR REPLACE] [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
VIEW `nombre_vista` [(`columnas`)]
AS `sentencia_select`
[WITH [CASCADED | LOCAL] CHECK OPTION]

Esta sentencia crea una vista nueva o reemplaza una existente si se incluye la cláusula `OR REPLACE`. La `sentencia_select` es una sentencia `SELECT` que proporciona la definición de la vista. Puede estar dirigida a tablas de la base o a otras vistas.

Se requiere que posea el permiso `CREATE VIEW` para la vista, y algún privilegio en cada columna seleccionada por la sentencia `SELECT`. Para columnas incluidas en otra parte de la sentencia `SELECT` debe poseer el privilegio `SELECT`. Si está presente la cláusula `OR REPLACE`, también deberá tenerse el privilegio `DELETE` para la vista.

Toda vista pertenece a una base de datos. Por defecto, las vistas se crean en la base de datos actual. Pera crear una vista en una base de datos específica, indíquela con `base_de_datos.nombre_vista` al momento de crearla.

```
mysql> CREATE VIEW test.v AS SELECT * FROM t;
```

Las tablas y las vistas comparten el mismo espacio de nombres en la base de datos, por eso, una base de datos no puede contener una tabla y una vista con el mismo nombre.

Al igual que las tablas, las vistas no pueden tener nombres de columnas duplicados. Por defecto, los nombres de las columnas devueltos por la sentencia `SELECT` se usan para las columnas de la vista . Para dar explícitamente un nombre a las columnas de la vista utilice la clásula *columnas* para indicar una lista de nombres separados con comas. La cantidad de nombres indicados en `columnas` debe ser igual a la cantidad de columnas devueltas por la sentencia `SELECT`.

Las columnas devueltas por la sentencia SELECT pueden ser simples referencias a columnas de la tabla, pero tambien pueden ser expresiones conteniendo funciones, constantes, operadores, etc.

Los nombres de tablas o vistas sin calificar en la sentencia SELECT se interpretan como pertenecientes a la base de datos actual. Una vista puede hacer referencia a tablas o vistas en otras bases de datos precediendo el nombre de la tabla o vista con el nombre de la base de datos apropiada.

Las vistas pueden crearse a partir de varios tipos de sentencias SELECT. Pueden hacer referencia a tablas o a otras vistas. Pueden usar combinaciones, UNION, y subconsultas. El SELECT inclusive no necesita hacer referencia a otras tablas. En el siguiente ejemplo se define una vista que selecciona dos columnas de otra tabla, así como una expresión calculada a partir de ellas:

```
mysql> CREATE TABLE t (qty INT, price INT);
mysql> INSERT INTO t VALUES(3, 50);
mysql> CREATE VIEW v AS SELECT qty, price, qty*price AS value FROM t;
mysql> SELECT * FROM v;
+------+-------+-------+
| qty  | price | value |
+------+-------+-------+
|    3 |    50 |   150 |
+------+-------+-------+
```

La definición de una vista está sujeta a las siguientes limitaciones:

- La sentencia SELECT no puede contener una subconsulta en su cláusula FROM.
- La sentencia SELECT no puede hacer referencia a variables del sistema o del usuario.
- La sentencia SELECT no puede hacer referencia a parámetros de sentencia preparados.
- Dentro de una rutina almacenada, la definición no puede hacer referencia a parámetros de la rutina o a variables locales.
- Cualquier tabla o vista referenciada por la definición debe existir. Sin embargo, es posible que después de crear una vista, se elimine alguna tabla o vista a la que se hace referencia. Para comprobar la definición de una vista en busca de problemas de este tipo, utilice la sentencia CHECK TABLE.
- La definición no puede hacer referencia a una tabla TEMPORARY, y tampoco se puede crear una vista TEMPORARY.
- Las tablas mencionadas en la definición de la vista deben existir siempre.
- No se puede asociar un disparador con una vista.

En la definición de una vista está permitido ORDER BY, pero es ignorado si se seleccionan columnas de una vista que tiene su propio ORDER BY.

Con respecto a otras opciones o cláusulas incluidas en la definición, las mismas se agregan a las opciones o cláusulas de cualquier sentencia que haga referencia a la vista creada, pero el efecto es indefinido. Por ejemplo, si la definición de una vista incluye una cláusula LIMIT, y se hace una selección desde la vista utilizando una sentencia que tiene su propia cláusula LIMIT, no está definido cuál se aplicará. El mismo principio se extiende a otras opciones como ALL, DISTINCT, o SQL_SMALL_RESULT que se ubican a continuación de la palabra reservada SELECT, y a cláusulas como INTO, FOR UPDATE, LOCK IN SHARE MODE, y PROCEDURE.

Si se crea una vista y luego se modifica el entorno de proceso de la consulta a traves de la modificación de variables del sistema, puede afectar los resultados devueltos por la vista:

```
mysql> CREATE VIEW v AS SELECT CHARSET(CHAR(65)), COLLATION(CHAR(65));
Query OK, 0 rows affected (0.00 sec)

mysql> SET NAMES 'latin1';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT * FROM v;
+------------------+--------------------+
| CHARSET(CHAR(65)) | COLLATION(CHAR(65)) |
+------------------+--------------------+
| latin1           | latin1_swedish_ci   |
+------------------+--------------------+
1 row in set (0.00 sec)

mysql> SET NAMES 'utf8';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM v;
+------------------+--------------------+
| CHARSET(CHAR(65)) | COLLATION(CHAR(65)) |
+------------------+--------------------+
| utf8             | utf8_general_ci     |
+------------------+--------------------+
1 row in set (0.00 sec)
```

La cláusula opcional ALGORITHM es una extensión de MySQL al SQL estándar. ALGORITHM puede tomar tres valores: MERGE, TEMPTABLE, o UNDEFINED. El algoritmo por defecto es UNDEFINED si no se encuentra presente la cláusula ALGORITHM. El algoritmo afecta la manera en que MySQL procesa la vista.

Para MERGE, el texto de una sentencia que haga referencia a la vista y la definición de la vista son mezclados de forma que parte de la definición de la vista reemplaza las partes correspondientes de la consulta.

Para TEMPTABLE, los resultados devueltos por la vista son colocados en una tabla temporal, la cual es luego utilizada para ejecutar la sentencia.

Para UNDEFINED, MySQL determina el algoritmo que utilizará. En ese caso se prefiere MERGE por sobre TEMPTABLE si es posible, ya que MERGE por lo general es más eficiente y porque la vista no puede ser actualizable si se emplea una tabla temporal.

Una razón para elegir explícitamente TEMPTABLE es que los bloqueos en tablas subyacentes pueden ser liberados despues que la tabla temporal fue creada, y antes de que sea usada para terminar el procesamiento de la sentencia. Esto podría resultar en una liberación del bloqueo más rápida que en el algoritmo MERGE, de modo que otros clientes que utilicen la vista no estarán bloqueados mucho tiempo.

El algoritmo de una vista puede ser UNDEFINED en tres situaciones:

- No se encuentra presente una cláusula ALGORITHMen la sentencia CREATE VIEW.
- La sentencia CREATE VIEW tiene explícitamente una cláusula ALGORITHM = UNDEFINED.
- Se especificó ALGORITHM = MERGE para una vista que solamente puede ser procesada usando una tabla temporal. En este caso, MySQL emite una advertencia y establece el algoritmo en UNDEFINED.

Como se dijo anteriormente, MERGE provoca que las partes correspondientes de la definición de la vista se combinen dentro de la sentencia que hace referencia a la vista. El siguiente ejemplo muestra brevemente cómo funciona el algoritmo MERGE. El ejemplo asume que hay una vista v_merge con esta definición:

```
CREATE ALGORITHM = MERGE VIEW v_merge (vc1, vc2) AS
```

SELECT c1, c2 FROM t WHERE c3 > 100;
Ejemplo 1: Suponiendo que se utilice esta sentencia:
SELECT * FROM v_merge;
MySQL la gestiona del siguiente modo:

- `v_merge` se convierte en `t`
- * se convierte en `vc1, vc2`, que corresponden a `c1, c2`
- Se agrega la cláusula `WHERE` de la vista

La sentencia ejecutada resulta ser:
SELECT c1, c2 FROM t WHERE c3 > 100;
Ejemplo 2: Suponiendo que se utilice esta sentencia:
SELECT * FROM v_merge WHERE vc1 < 100;

Esta sentencia se gestiona en forma similar a la anterior, a excepción de que `vc1 < 100` se convierte en `c1 < 100` y la cláusula `WHERE` de la vista se agrega a la cláusula `WHERE` de la sentencia empleando un conector `AND` (y se agregan paréntesis para asegurarse que las partes de la cláusula se ejecutarán en el orden de precedencia correcto). La sentencia ejecutada resulta ser:
SELECT c1, c2 FROM t WHERE (c3 > 100) AND (c1 < 100);
Necesariamente, la sentencia a ejecutar tiene una cláusula `WHERE` con esta forma:
WHERE (WHERE de la sentencia) AND (WHERE de la vista)

El algoritmo `MERGE` necesita una relación uno-a-uno entre los registros de la vista y los registros de la tabla subyacente. Si esta relación no se sostiene, debe emplear una tabla temporal en su lugar. No se tendrá una relación uno-a-uno si la vista contiene cualquiera de estos elementos:

- Funciones agregadas (`SUM()`, `MIN()`, `MAX()`, `COUNT()`, etcétera)
- `DISTINCT`
- `GROUP BY`
- `HAVING`
- `UNION` o `UNION ALL`
- Hace referencia solamente a valores literales (en tal caso, no hay una tabla subyacente)

Algunas vistas son actualizables. Esto significa que se las puede emplear en sentencias como `UPDATE`, `DELETE`, o `INSERT` para actualizar el contenido de la tabla subyacente. Para que una vista sea actualizable, debe haber una relación uno-a-uno entre los registros de la vista y los registros de la tabla subyacente. Hay otros elementos que impiden que una vista sea actualizable. Más específicamente, una vista no será actualizable si contiene:

- Funciones agregadas (`SUM()`, `MIN()`, `MAX()`, `COUNT()`, etcétera)
- `DISTINCT`
- `GROUP BY`
- `HAVING`
- `UNION` o `UNION ALL`
- Una subconsulta en la lista de columnas del SELECT
- Join
- Una vista no actualizable en la cláusula `FROM`
- Una subconsulta en la cláusula `WHERE` que hace referencia a una tabla en la cláusula `FROM`
- Hace referencia solamente a valores literales (en tal caso no hay una) tabla subyacenta para actualizar.
- `ALGORITHM = TEMPTABLE` (utilizar una tabla temporal siempre resulta en una vista no actualizable)

Con respecto a la posibilidad de agregar registros mediante sentencias `INSERT`, es necesario que las columnas de la vista actualizable también cumplan los siguientes requisitos adicionales:

- No debe haber nombres duplicados entre las columnas de la vista.

- La vista debe contemplar todas las columnas de la tabla en la base de datos que no tengan indicado un valor por defecto.
- Las columnas de la vista deben ser referencias a columnas simples y no columnas derivadas. Una columna derivada es una que deriva de una expresión. Estos son algunos ejemplos de columnas derivadas:
  3.14159
  col1 + 3
  UPPER(col2)
  col3 / col4
  (subquery)

No puede insertar registros en una vista conteniendo una combinación de columnas simples y derivadas, pero puede actualizarla si actualiza únicamente las columnas no derivadas. Considere esta vista:
CREATE VIEW v AS SELECT col1, 1 AS col2 FROM t;

En esta vista no pueden agregarse registros porque col2 es derivada de una expresión. Pero será actualizable si no intenta actualizar col2. Esta actualización es posible:
UPDATE v SET col1 = 0;

Esta actualización no es posible porque se intenta realizar sobre una columna derivada:
UPDATE v SET col2 = 0;

A veces, es posible que una vista compuesta por múltiples tablas sea actualizable, asumiendo que es procesada con el algoritmo MERGE. Para que esto funcione, la vista debe usar inner join (no outer join o UNION). Además, solamente puede actualizarse una tabla de la definición de la vista, de forma que la cláusula SET debe contener columnas de sólo una tabla de la vista. Las vistas que utilizan UNION ALL no se pueden actualizar aunque teóricamente fuese posible hacerlo, debido a que en la implementación se emplean tablas temporales para procesarlas.

En vistas compuestas por múltiples tablas, INSERT funcionará si se aplica sobre una única tabla. DELETE no está soportado.
La cláusula WITH CHECK OPTION puede utilizarse en una vista actualizable para evitar inserciones o actualizaciones excepto en los registros en que la cláusula WHERE de la *sentencia_select* se evalúe como true.

En la cláusula WITH CHECK OPTION de una vista actualizable, las palabras reservadas LOCAL y CASCADED determinan el alcance de la verificación cuando la vista está definida en términos de otras vistas. LOCAL restringe el CHECK OPTION sólo a la vista que está siendo definida. CASCADED provoca que las vistas subyacentes también sean verificadas. Si no se indica, el valor por defecto es CASCADED. Considere las siguientes definiciones de tabla y vistas:

```
mysql> CREATE TABLE t1 (a INT);
mysql> CREATE VIEW v1 AS SELECT * FROM t1 WHERE a < 2
   -> WITH CHECK OPTION;
mysql> CREATE VIEW v2 AS SELECT * FROM v1 WHERE a > 0
   -> WITH LOCAL CHECK OPTION;
mysql> CREATE VIEW v3 AS SELECT * FROM v1 WHERE a > 0
   -> WITH CASCADED CHECK OPTION;
```

Las vistas v2 y v3 estan definidas en términos de otra vista, v1. v2 emplea check option LOCAL, por lo que las inserciones sólo atraviesan la verificación de v2 . v3 emplea check option CASCADED de modo que las inserciones no solamente atraviesan su propia verificación sino tambien las de las vistas subyacentes. Las siguientes sentencias demuestran las diferencias:

```
ql> INSERT INTO v2 VALUES (2);
```

```
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO v3 VALUES (2);
ERROR 1369 (HY000): CHECK OPTION failed 'test.v3'
```

La posibilidad de actualización de las vistas puede verse afectada por el valor de la variable del sistema `updatable_views_with_limit`.

La sentencia `CREATE VIEW` fue introducida en MySQL 5.0.1. La cláusula `WITH CHECK OPTION` fue implementada en MySQL 5.0.2.

`INFORMATION_SCHEMA` contiene una tabla `VIEWS` de la cual puede obtenerse información sobre los objetos de las vistas.

10.2.1 Sintaxis de `DROP VIEW`

DROP VIEW [IF EXISTS]
  nombre_vista [, `nombre_vista`] ...
  [RESTRICT | CASCADE]

`DROP VIEW` elimina una o más vistas de la base de datos. Se debe poseer el privilegio `DROP` en cada vista a eliminar.

La cláusula `IF EXISTS` se emplea para evitar que ocurra un error por intentar eliminar una vista inexistente. Cuando se utiliza esta cláusula, se genera una `NOTE` por cada vista inexistente.

`RESTRICT` y `CASCADE` son ignoradas.

10.2.2 Sintaxis de `SHOW CREATE VIEW`

SHOW CREATE VIEW `nombre_vista`

Muestra la sentencia `CREATE VIEW` que se utilizó para crear la vista.

```
mysql> SHOW CREATE VIEW v;
+-------+-----------------------------------------------------+
| Table | Create Table                                        |
+-------+-----------------------------------------------------+
| v     | CREATE VIEW `test`.`v` AS select 1 AS `a`,2 AS `b` |
+-------+-----------------------------------------------------+
```

11 Copia de seguridad

11.1 El programa de copia de seguridad de base de datos *mysqldump*

El cliente **mysqldump** puede utilizarse para volcar una base de datos o colección de bases de datos para copia de seguridad o para transferir datos a otro servidor SQL (no necesariamente un servidor MySQL). EL volcado contiene comandos SQL para crear la tabla y/o rellenarla.

Si está haciendo una copia de seguridad del servidor, y las tablas son todas `MyISAM`, puede considerar usar **mysqlhotcopy** ya que hace copias de seguridad más rápidas y restauraciones más rápidas, que pueden realizarse con el segundo programa.

Hay tres formas de invocar **mysqldump**:
shell> mysqldump [*opciones*] `nombre_de_base_de_datos` [`tablas`]
shell> mysqldump [*opciones*] --databases `DB1` [*DB2* `DB3`...]
shell> mysqldump [*opciones*] –all-databases

Si no se nombra ninguna tabla o se utiliza la opción `--databases` o `--all-databases`, se vuelca bases de datos enteras.

Para obtener una lista de las opciones que soporta su versión de **mysqldump**, ejecute **mysqldump –help**.

Para respaldar la base de datos existente:
```
shell> mysqldump -uussuario -pclave -R nombre_base_de_datos >
nombre_del_archivo_de_respaldo
```

11.2 Mysqladmin para borrar y crear bases de datos

mysqladmin es un cliente para realizar operaciones administrativas. Se puede usar para comprobar la configuración y el estado actual del servidor, crear y borrar bases de datos, y con más finalidades.

Invoque **mysqladmin** así:
shell> mysqladmin [`opciones`] *comando* [`opciones_de_comando`] *comando* ...

mysqladmin soporta los siguientes comandos:

- `create` *nombre_base_de_datos*
  Crea una nueva base de datos llamada *nombre_base_de_datos*.
- `drop` *nombre_base_de_datos*
  Borra la base de datos llamada `nombre_base_de_datos` y todas sus
- Muchas mas opciones. Ver la documentación de MySQL en el CD del curso.

Para borrar la base de datos existente:
```
shell> mysqladmin -uussuario -pclave drop nombre_base_de_datos
```

Para crear una base de datos vacilla:
```
shell> mysqladmin -uussuario -pclave create nombre_base_de_datos
```


11.3 Mysql para llenar una base de datos

mysql es un simple shell SQL (con capacidades GNU readline). Soporta uso interactivo y no interactivo. Cuando se usa interactivamente, los resultados de las consultas se muestran en formato de tabla ASCII. Cuando se usa no interactivamente (por ejemplo, como filtro), el resultado se presenta en formato separado por tabuladores. El formato de salida puede cambiarse usando opciones de línea de comandos.

Si tiene problemas relacionados con memoria insuficiente para conjuntos de resultados grandes, utilice la opción `--quick`. Esto fuerza **mysql** a devolver los resultados desde el servidor registro a registro en lugar de recibir todo el conjunto de resultados y almacenarlo en memoria antes de mostrarlo. Esto se hace usando `mysql_use_result()` en lugar de `mysql_store_result()` para recibir el conjunto de resultados.

Usar **mysql** es muy sencillo. Invóquelo desde el prompt de su intérprete de comandos como se muestra a continuación:
shell> mysql `nombre_base_de_datos`

```
Para rellenar la base de datos con el archivo de respaldo ejectua:
shell> mysql -uussuario -pclave -f nombre_base_de_datos > archivo_de_respaldo
```


12 Ejercicios

1. Crear una tabla con columnas de los diferente tipos numericos, y probar de insertar filas con valores dentro y luego fuera de los rangos minimo y maximo de cada tipo.
2. Crear una tabla con columnas de los diferente tipos de fechas, y probar de insertar filas con valores dentro y luego fuera de los rangos minimo y maximo de cada tipo.
3. Crear una tabla con columnas de los diferente tipos de caracteres, y probar de insertar filas con valores para cada tipo.
4. Probar las precedencia de operadores
5. Probar las funciones de comparacion de datos con enteros.
6. Probar los operdores logicos
7. Probar las funciones de flujo
8. Probar las funciones de comparacion de datos con caracteres.
9. Probar las funciones de matematica
10. Probar las funciones de fechas.
11. Probar las funciones de encriptacion.
12. Probar las funciones de información
13. Crear un procedimiento para crear un usuario
14. Crear un procedimiento que regresa el numero de registros que tiene la tabla lab_prescription
15. Crear un procedimiento para crear un paciente, y que regresa el no_patient generado, o un numero negativo si el no_lnaiss o el no_city no existe en la tabla city.
16. Crear un procedimiento que va a usar un cursor y poner inactivo todos los usuarios que tienen el campo bad_try >1
17. Crear una funcion que regresa el nombre y apellido completo del paciente cual no_patient fue pasado como parametro.
18. Crear una funcion que va a regresar la edad un paciente.
19. Crear una funcion que regresa el numero de lab_prescription ligadas a un paciente
20. Crear una funcion que regresa el numero de pruebas (lab_test) de un cierto tipo (lab_test_type) ligadas a un paciente
21. Crear una funcion que devuelve un numero unico y que se va incrementando (como una secuencia).
22. Crear un trigger que va a poner el mayusculas los nombres y apellidos del paciente al insertar o modificar el paciente.
23. Crear un trigger que va a crear un usuario al insertar un paciente, con el mismo nombre/apellido.
24. Crear una vista que contiene los datos del paciente asi que los nombre de cuidades de nacimiento y residencia.
25. Crear una vista que lista los pacientes por region y cuidad, y da el total de pacientes por cada uno.
26. Crear una copia de su base de datos usando respaldo/restauracion de su base de datos.