6 MySQL Referencia del lenguage

MySQL dispone de una interficie muy compleja pero intuitiva. Este capítulo describe los diferentes comandos, tipos de datos, y funciones que necesitarás conocer en el momento de utilizar MySQL eficientemente y efectivamente. Este capítulo también sirve de referencia para todas las funcionalidades incluídas en MySQL. Con la intención de utilizar este capítulo de una manera efectiva, puedes encontrar útil ver los diferentes apartados del índice.

6.1. Estructura del lenguage

6.1.1. Literales: cómo escribir cadenas y números

Esta sección describe las diferentes maneras de escribir cadenas y números en MySQL. Ello también cubre los múltiples matices y [gotchas] que puedes utilizar al tratar con los tipos de datos MySQL.

6.1.1.1. Cadenas

Una cadena es una secuencia de caracteres, enmarcados por comillas simples (‘), o comillas dobles(“), aunque en modo ANSI sólo se utiliza la comilla simple. Ejemplos:

```
‘a string’
“another string”
```

Dentro de una cadena, ciertas secuencias tienen un significado especial. Cada una de estas secuencias empieza por una contrabarra (‘\’), conocido como carácter de escape. MySQL reconoce las siguientes secuencias de escape:

Secuencia	Significado
\0	Carácter nulo de ASCII (NULL)
\’	Carácter de comilla simple.
\”	Caracter de comillas dobles.
\b	Un carácter de borrado (backspace)
\n	Carácter de nueva línea
\r	Carácter de retorno de carro
\t	Carácter de tabulación
\z	ASCII 26 (Control-Z). Este carácter es particularmente importante, porque indica el final de archivo (END-OF-FILE) en Windows., de modo que podría causar problemas en caso de utilizar comandos del tipo: `use mysql database <filename`
\\	Carácter de contrabarra
\%	Carácter ‘%’. Habitualmente es usuado como búsqueda literal de este carácter en entornos donde el símbolo porcentual se interpreta como un comodín. Puedes ver la sección 6.3.2.1 [String comparison functions]
_	Como en el caso anterior, el subrallado literal se utiliza para diferenciarse del uso habitual como comodín. Puedes ver la sección 6.3.2.1 [String comparison functions]

Cabe anotar que en los dos últimos casos, y en ciertos contextos de cadenas, se retornará ‘\%’ y ‘_’, y no ‘%’ y ‘_’.

Hay varias maneras de incluir comillas simples dentro de una cadena:

- Una ‘ dentro de una cadena enmarcada con comillas simples puede ser escrita como ‘ ‘.
- Una “ dentro de una cadena enmarcada con comillas dobles puede ser escrita como “ “.
- Puedes preceder el carácter de comillas con un carácter de escape (\).
- Las comillas simples dentro de una cadena enmarcada con comillas dobles no necesita un tratamiento especial y no necesita ser doblada o marcada con caracteres de escape. Del mismo modo, las comillas dobles dentro de comillas simples tampoco necesitan tratamientos especiales.

Las sentencias SELECT presentados aquí demuestran el modo en el que entrecomillado y las secuencias de escape funcionan:

```
mysql> SELECT ‘hello’, ‘”hello”’, ‘””hello””’, ‘hel’’lo’,’\’hello’;
```

Resultado:

```
hello, “hello”, “”hello””, hel’lo ‘hello

mysql> SELECT “This\nIs\nFour\Lines”;

This
Is
Four
Lines
```

Si quieres insertar datos binarios en un campo de cadena (como BLOB), los siguientes caracteres deben ser representados por secuencias de escape:

Carácter	Representación
NUL	ASCII 0. Deberías representarlo con \0
\	ASCII 92, o contrabarra. Se representa con \\
‘	ASCII 39, comillas simples. Se representa con \’
“	ASCII 34, comillas dobles. Se representan con \”

Si escribes en código C, puedes utilizar la función de API de C mysql_real_scape_string() para los caracteres de escape utilizados en una sentencia INSERT. Puedes ver la sección 8.4.2 [C API function overview]. Con Perl, puedes utilizar el método quote del paquete DBI para convertir los caracteres especiales en secuencias de escape adecuadas. Puedes ver la sección 8.2.2 [Perl DBI Class].

Deberías utilizar una secuencia de escape en cualquier cadena donde aparezcan los caracteres indicados arriba.

Como alternativa, varias API’s de MySQL aportan la posibilidad de insertar marcadores especiales en la cadena de una consulta, y entonces asociar los valores de los datos a ellos cuando utilices cada consulta. En este caso, la API tiene en cuenta por ti los caracteres especiales automáticamente.

6.1.1.2. Números

Los enteros son representados como una secuencia de dígitos. Los flotantes usan ‘.’ como separador decimal. Cualquier tipo de número puede ir precedido por un signo ‘- ’ para indicar un valor negativo.

Ejemplos de enteros válidos:
1221
0
-32

Ejempos de valores flotantes válidos:

294.42
32032.6809e+10
148.00

Un entero puede ser usado en un contexto de valores de coma flotante; éste es interpretado como el equivalente en valor de coma flotante.

6.1.1.3. Valores hexadecimales

MySQL soporta valores hexadecimales. En un contexto numérico actúan como un entero (con precisión de 64 bits). En un contexto de cadenas, éstos actúan como una cadena binaria donde cada par de dígitos hexadecimales es convertido en carácter.

```
mysql> SELECT x'FF';
                ->255
mysql> SELECT 0xa+0;
                ->10
mysql> SELECT 0x5061756c;
                ->Paul
```

La sintaxis x'cadena_hexadecimal' (nueva en 4.0) está basada en el ANSI SQL y la sintaxis 0x está basada en ODBC. Las cadenas hexadecimales son usadas habitualmente en ODBC para proveer valores en campos BLOB. Puedes convertir una cadena o un número a hexadecimal con la función HEX().

6.1.1.4. Valores NULL

El valor NULL significa "no hay datos", y es diferente de los valores como 0 para los tipos numéricos o la cadena vacía. Puedes ver la sección A.5.3. [Problems with NULL].

El valor NULL puede ser representado po \N cuando se utiliza la importación/exportación de archivos de texto (LOAD DATA INFILE, SELECT ... INTO OUTFILE). Puedes ver la sección 6.4.9.[LOAD DATA].

6.1.2. Base de datos, índice, columna (campo), y Alias

Base de datos, índice, columna (campo), y Alias, todos ellos siguen las mismas reglas en MySQL. Cabe indicar que las reglas cambiaron a partir de la versión 3.23.6 de MySQL, cuando se introdujo el entrecomillado de los identificadores (bases de datos, tablas y nombres de campos) con comillas simples. Las comillas dobles tendrán efecto también para entrecomillar identificadores al funcionar en modo ANSI. Puedes ver la sección 1.7.2 [modo ANSI].

Identificador	Máxima longitud	Caracteres permitidos

Database	64	Cualquier carácter permitido en un nombre de directorio, exceptuando /, \, y '.'
Table	64	Cualquier carácter permitido para un directorio excepto /, o '.'
Column	64	Todos los caracteres
Alias	255	Todos los caracteres

Además de lo anterior, no se puede utilizar ASCII(0) o ASCII(255) o el carácter de comillas en un identificador.

Cabe anotar que si el identificador es una palabra reservada o contiene caracteres especiales, debes entrecomillarlo siempre con ` cuando lo utilices:

```
mysql> SELECT * FROM `select` WHERE `select`.id > 100;
```

Puedes ver la sección 6.1.7. [palabras reservadas]

En las versiones de MySQL anteriores a 3.23.6, las reglas de nombres eran como siguen:

- Un nombre puede consistir de caracteres alfanuméricos del juego de caracteres utilizado, y también _ y '$'. El juego de caracteres por defecto es el ISO-8859-1 Latin1; Puede ser cambiado con la opción de –-default-character-set to mysqld. Puedes ver la sección 4.6.1 [Juegos de caracteres].
- Un nombre puede empezar con cualquier carácter que es legal en un nombre. En particular, un nombre puede empezar con dígito (¡esto difiere de la mayoría de sistemas de bases de datos!). Sin embargo, un nombre no puede consistir sólo en dígitos.
- No puedes utilizar el carácter '.' en nombres porque se utiliza para la separación entre los identificadores de tablas y columnas [De la forma: tabla.columna].

Se recomienda que no uses nombres como 1e, debido a que una expresión como 1e+1 es ambigua. Puede ser interpretada como la expresión 1e+1 o como el nombre 1e+1.

En MySQL te puedes referir a una columna utilizando cualquiera de las formas siguientes:

Referencia a columna	Significado
nomb_col	Columna nomb_col, de la tabla utilizada en la consulta.
nomb_tabl.nomb_col	columna nomb_col de la tabla nomb_tabl de la base de datos actual
nomb_db.nomb_tabl.nomb_col	Columna nomb_col de la tabla nomb_tabl de la base de datos db_name. Este formato está disponible en la versión 3.22 y superiores
` nomb_columna`	Columna clave o contiene caracteres especiales.

No necesitas especificar el nombre de la tabla o de la base de datos para referirte a una columna en una sentencia, a no ser que la referencia pueda ser ambigua. Por ejemplo, supón que las tablas t1 y t2 contienen una columna c, y quieres recuperar c en una sentencia SELECT que utiliza t1 y t2. En este caso, c es ambigua ya que no es única en las tablas a las que se refiere la sentencia, por lo que debes indicar en qué tabla te refieres escribiendo t1.c, ó t2.c. De un modo similar, si estás recuperando datos de la tabla t en la base de datos db1 y de la tabla t en la base de datos db2, debes referirte a las columnas de esta tabla como db1.t.nomb_col y db2.t.nomb_col

La sintaxis .nomb_tabl significa que la tabla nom_tabl se halla en la base de datos actual. Esta sintaxis es aceptada por compatibilidad con ODBC, debido a que algunos programas ODBC prefijan los nombres de tablas con un carácter ‘ . ’ .

6.1.3. Diferenciación mayúsculas/minúsculas (Case sensitivity) en los nombres

En MySQL las bases de datos y tablas se corresponden con directorios y ficheros dentro de tales directorios. Consecuentemente, la diferenciación entre mayúsculas y minúsculas del sistema operativo que soporte a MySQL determina este aspecto. En el caso de Windows, no se diferencia entre mayúsculas y minúsculas, mientras que en la mayoría de sistemas UNIX (con la excepción de Mac OS X) se diferencia entre mayúsculas y minúsculas.

Nota: Aunque en Windows no se diferencie entre mayúsculas y minúsculas, es conveniente no referirse en una consulta a la misma tabla con diferentes combinaciones de mayúsculas y minúsculas. La siguiente consulta podría no funcionar debido a este hecho:

```
mysql> SELECT * FROM my_table WHERE MY_TABLE.col=1;
```

Los nombres de columnas y sus alias no diferencian en ningún caso mayúsc/minúsc.

Los Alias de las tablas también son case-sensitive. La siguiente consulta podría no funcionar porque se refiere al alias con a y A:

```
mysql> SELECT nomb_col FROM nomb_tabl AS a
                WHERE a.nomb_col=1 OR A.nomb_col=2;
```

Si tienes problemas recordando la combinación de mayúsculas y minúsculas utilizadas en tus tablas, adopta una convención consistente, como por ejemplo crear nombres de tablas y bases de datos en minúsculas.

Una forma de evitar este problema es iniciar pysqld con –0 lower_case_table_names=1. Por defecto esta opción es 1 en Windows y 0 en Unix.

Si lower_case_table_names es 1 MySQOL convertirá todos los nombres de tablas a minúsculas en almacenamiento y presentación [lookup]. Cabe notar que si cambias esta opción, deberás convertir todos los nombres de tus tablas a minúsculas antes de iniciar mysqld.

Si mueves archivos MyISAM de Windows a un disco *nix, puedes necesitas el uso de mysql_fix_extensions en algunos casos, herramienta para amañar mayusc/minusc de las extensiones de los ficheros en cada base de datos especificada en cada directorio (minúsculas ‘.frm’, mayúsculas en ‘.MYI’ y ‘.MYD’). mysql_fix_extensions puede ser encontrado en directorio script.

6.1.4. Variables del usuario

MySQL soporta variables de usuario específicas de conexiones con la sintaxis @variablename. Un nombre de variable puede consistir en caracteres alfanuméricos propios del juego de caracteres por defecto, y también ‘_’, ‘$’ y ‘.’. El valor por defecto es ISO-8859-1 Latin1, que puede ser cambiado por la opción –default-character-set en mysqld. Puedes ver la sección 4.6.1. [Character Sets].

Las variables no precisan ser inicializadas. Contienen NULL por defecto y pueden almacenar un valor entero, real, o una cadena. Todas las variables de una ejecución [Thread] son liberadas automáticamente cuando ésta acaba.

Se puede crear una variable con la sintaxis SET:

SET @variable = {valor entero | valor real | cadena}
[,@variable2=...].

Puedes asignar también un valor a una variable en otras sentencias diferentes de SET. Sin embargo, en este caso el operador de asignación es := en vez de =, ya que = se reserva para comparaciones en sentencias diferentes de SET:

mysql> SELECT @t1:=(@t2:=1)+@t3:=4,@t1,@t2,@t3;

@t1:=(@t2:=1)+@t3:=4	@t1	@t2	@t3
5	5	1	4

Las variables de usuario pueden ser utilizadas donde se permiten expresiones. Cabe notar que ello no incluye habitualmente contextos donde el número es requerido explícitamente, como por ejemplo en el caso de la cláusula LIMIT de una sentencia SELECT, o la cláusula IGNORE <valor> LINES de la sentencia LOAD DATA.

Nota: en una sentencia SELECT, cada expresión es evaluada sólo cuando es enviada al cliente. Esto significa que en una cláusula como HAVING, GROUP BY, o ORDER BY, no te puedes referir a una expresión que envuelva variables indicadas en el apartado SELECT. Por ejemplo, la siguiente sentencia NO funcionará como se espera:

mysql> SELECT (@aa:=id) AS a, (@aa+3) AS b FROM nomb_tabla HAVING b=5;

La razón es que la @aa no contendrá el valor de la fila actual, sino el valor de id por la anterior fila aceptada.

6.1.5. Variables del sistema

A partir de la versión 4.0.3 aportamos mejor acceso a un gran número de variables del sistema y de conexión. Podemos cambiar la mayoría de ellas sin tener que parar el servidor.

Hay dos tipos de variables del sistema: Variables específicas de ejecuciones [Thread] (o conexiones) que se refieren únicamente a la conexión actual, y variables globales que se utilizan para configurar eventos globales o como variables iniciales en una nueva conexión.

Cuando mysqld se inicia, todas las variables globales se inicializan desde los argumentos de la línea de comandos y los archivos de opciones. Puedes cambiar los valores usados con el comando SET GLOBAL. Cuando una se crea nueva ejecución, sus variables específicas son inicializadas a partir de variables globales y no varían, aunque se utilice una sentencia SET GLOBAL.

Para disponer el valor de una variable GLOBAL, debes usar una de las siguientes sintaxis (aquí utilizamos sort_buffer_size como una variable de ejemplo):

SET GLOBAL sort_buffer_size=value;
SET @@global.sort_buffer_size=value;

Para disponer el valor para la variable SESSION, puedes utilizar una de las siguientes sintaxis:

SET SESSION sort_buffer_size=value;
SET @@session.sort_buffer_size=value;
SET sort_buffer_size=value;

Si no especificas las cláusulas GLOBAL o SESSION, el sistema utilizará SESSION. Puedes ver la sección 5.5.6 [SET OPTION].

LOCAL es un sinónimo de SESSION.

Para recuperar el valor de una variable global puedes utilizar uno de los siguientes comandos:

SELECT @@global.sort_buffer_size;
SHOW SESSION VARIABLES like 'sort_buffer_size';

Cuando recuperas el valor de una variable con la sintaxis @@variable_name sin especificar GLOBAL o SESSION, entonces MySQL retorn el valor específico de la ejecución (SESSION), si existe. Si no es así, MySQL retornará el valor global.

La razón para requerir GLOBAL para disponer variables GLOBAL pero no para simplemente recuperar sus valores es asegurar que no provocaremos problemas en usos posteriores al introducir una variable específica con el mismo nombre, o borrar una variable específica de la ejecución. En este caso podrías cambiar accidentalemente el estado para todo el servidor, y no sólo para tu conexión. A continuación puedes ver una lista completa de todas las variables que puedes usar y cambiar, usando GLOBAL o SESSION con ellas:

Nombre de la variable	Tipo de valor	Tipo
autocommit	bool	SESSION
big_tables	bool	SESSION
binlog_cache_size	num	GLOBAL
bulk_insert_buffer_size	num	GLOBAL/SESSION
concurrent_insert	bool	GLOBAL
connect_timeout	num	GLOBAL
convert_character_set	string	SESSION
delay_key_write	bool	GLOBAL
delayed_insert_limit	num	GLOBAL
delayed_insert_timeout	num	GLOBAL
delayed_queue_size	num	GLOBAL
flush	bool	GLOBAL
flush_time	num	GLOBAL
identity	num	SESSION
insert_id	bool	SESSION
interactive_timeout	num	GLOBAL/ SESSION
join_buffer_size	num	GLOBAL/ SESSION
key_buffer_size	num	GLOBAL
last_insert_id	bool	SESSION
local_infile	bool	GLOBAL
log_warnings	bool	GLOBAL
long_query_time	num	GLOBAL/SESSION
low_priority_updates	bool	GLOBAL/SESSION
max_allowed_packet	num	GLOBAL/SESSION
max_binlog_cache_size	num	GLOBAL

max_binlog_size	num	GLOBAL
max_connect_errors	num	GLOBAL
max_connections	num	GLOBAL
max_delayed_threads	num	GLOBAL
max_heap_table_size	num	GLOBAL/SESSION
max_join_size	num	GLOBAL/SESSION
max_sort_length	num	GLOBAL/SESSION
max_tmp_tables	num	GLOBAL
max_user_connections	num	GLOBAL
max_write_lock_count	num	GLOBAL
myisam_max_extra_sort_file_size	num	GLOBAL/SESSION
myisam_max_sort_file_size	num	GLOBAL/SESSION
myisam_soft_buffer_size	num	GLOBAL/SESSION
myisam_buffer_length	num	GLOBAL/SESSION
net_buffer_length	num	GLOBAL/SESSION
net_read_timeout	num	GLOBAL/SESSION
net_write_timeout	num	GLOBAL/SESSION
query_cache_limit	num	GLOBAL
query_cache_size	num	GLOBAL
query_cache_type	enum	GLOBAL
read_buffer_size	num	GLOBAL/SESSION
read_rnd_buffer_size	num	GLOBAL/SESSION
rpl_recovery_rank	num	GLOBAL
safe_show_database	bool	GLOBAL
server_id	num	GLOBAL
slave_net_timeout	num	GLOBAL
slow_launch_time	num	GLOBAL
sort_buffer_size	num	GLOBAL/SESSION
sql_auto_is_null	bool	SESSION
sql_big_selects	bool	SESSION
sql_big_tables	bool	SESSION
sql_buffer_result	bool	SESSION
sql_log_binlog	bool	SESSION
sql_log_off	bool	SESSION
sql_log_update	bool	SESSION
sql_low_priority_updates	bool	GLOBAL/SESSION
sql_max_join_size	num	GLOBAL/SESSION
sql_quote_show_create	bool	SESSION
sql_safe_updates	bool	SESSION
sql_select_limit	bool	SESSION
sql_slave_skip_counter	num	GLOBAL
sql_warnings	bool	SESSION
table_cache	num	GLOBAL
table_type	enum	GLOBAL/SESSION
thread_cache_size	num	GLOBAL
timestamp	bool	SESSION
tmp_table_size	enum	GLOBAL/SESSION
tx_isolation	enum	GLOBAL/SESSION
version	string	GLOBAL
wait_timeout	num	GLOBAL/SESSION

Las variables en las que el tipo de valor es num, pueden ser dadas por un valor numérico. Las variables en las que se indica bool pueden ser indicadas con un 0 o un 1, ON o OFF. Las

variables de tipo enum [enumeración] deberían contener uno de los valores disponibles para la variable, pero también pueden tener valores numéricos correspondientes al orden de la enumeración (el primer valor de la enumeración es 0).

Aquí podemos encontrar la descripción de algunas de las variables:

Variable	Descripción
identity	Alias para last_insert_id (compatibilidad Sybase)
sql_low_priority_updates	Alias para low_priority_updates
sq_max_join_size	Alias para max_join_size
version	Alias para VERSION() (compatibilidad Sybase (?))

La descripción de las definiciones de otras variables pueden hallarse en la sección de las opciones de arranque. La descripción de SHOW VARIABLES y en la sección de SET. Puedes ver la sección 4.1.1. [command-line options], la sección 4.5.6.4 [SHOW VARIABLES] y la sección 5.5.6. [SET OPTION].

6.1.6. Sintaxis de comentarios

El servidor MySQL soporta el # al final de línea, -- al final de línea y el /* ... */ para comentarios de una o más líneas Los estilos comentarios son:

```
mysql> SELECT 1+1; # este comentario llega hasta el final de la línea
mysql> SELECT 1+1; -- este comentario llega hasta el final de la línea
mysql> SELECT 1 /*este es un comentario in-line */+1;
mysql> SELECT 1+
/*
este es un comentario
de múltiples líneas*/
1;
```

¡Cabe anotar que el doble guión requiere que dejes al menos un espacio después del segundo guión! [posiblemente para evitar ambigüidades con los signos negativos].

A pesar de que el servidor entiende la sintaxis del comentario [just described], existen ciertas limitaciones en la forma con la que el cliente mysql interpreta comentarios del tipo /* ... */

- Caracteres de comillas simples y dobles son tomados para indicar el inicio de una cadena entrecomillada, incluso dentro de un comentario. Si las primeras comillas no es cerrada por unas segundas comillas, el intérprete no identificará el final del comentario. Si estás utilizando mysql interactivamente, puedes comprobar esta confusión al ver que la línea de comandos cambia de mysql a '> ó ">.
- Un punto y coma se utiliza para indicar el final de línea en la actual sentencia SQL, y cualquier elemento posterior a éste indica una nueva sentencia.

Estas limitaciones son aplicables tanto en la ejecución interactiva de MySQL como cuando se ejecutan los comandos desde un archivo, desde el cual se pide a mysql que los lea utilizando mysql < archivo.ext.

MySQL soporta el comentario -– de ANSI SQL sólo si el segundo guión del comentario va seguido de un espacio. Puedes ver la sección 1.7.4.7 [ANSI diff comments].

6.1.7. ¿MySQL es quisquillosa [Picky] con las palabras reservadas?

Un problema común aparece al tratar de crear una tabla con una columna cuyo nombre utiliza los nombres de tipos de datos o funciones propias de MySQL, como TIMESTAMP o GROUP. Tienes permiso para hacerlo (por ejemplo, ABS es un nombre permitido para una columna), pero el espacio en blanco no está permitido entre el nombre de una función y los paréntesis que incluyen los parámetros, en los casos en los que se utilicen tales nombres para nombres de columnas de tablas.

Las siguientes palabras están explícitamente reservadas en MySQL. Varias de ellas están prohibidas por el ANSI SQL92 como nombres de columnas y/o tablas (por ejemplo, GROUP). Algunas de ellas están reservadas porque MySQL las necesita y utiliza actualmente el intérprete yacc:

Palabra	Palabra	Palabra	Palabra
ADD	ALL	ALTER	ANALYZE
AND	AS	ASC	AUTO_INCREMENT
BDB	NERKELEYDB	BETWEEN	BIGINT
BINARY	BLOB	BOTH	BY
CASCADE	CASE	CHANGE	CHAR
CHARACTER	COLUMN	COLUMNS	CONSTRAINT
CREATE	CROSS	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	DATABASE	DATABASES	DAY_HOUR
DAY_MINUTE	DAY_SECOND	DEC	DECIMAL
DEFAULT	DELAYED	DELETE	DESC
DESCRIBE	DISTINCT	DISTINCTROW	DOUBLE
DROP	ELSE	ENCLOSED	ESCAPED
EXISTS	EXPLAIN	FIELDS	FLOAT
FOR	FOREIGN	FROM	FULLTEXT
FUNCTION	GRANT	GROUP	HAVING
HIGH_PRIORITY	HOUR_MINUTE	HOUR_SECOND	IF
IGNORE	IN	INDEX	INFILE
INNER	INNODB	INSERT	INT
INTEGER	INTERVAL	INTO	IS
JOIN	KEY	KEYS	KILL
LEADING	LEFT	LIKE	LIMIT
LINES	LOAD	LOCK	LONG
LONGBLOB	LONGTEXT	LOW_PRIORITY	MASTER_SERVER_ID
MATCH	MEDIUMBLOB	MEDIUMINT	MEDIUMTEXT
MIDDLEINT	MINUTE_SECOND	MRG_MYISAM	NATURAL
NOT	NUULL	NUMERIC	ON
OPTIMIZE	OPTION	OPTIONALLY	OR
ORDER	OUTER	OUTFILE	PARTIAL
PRECISSION	PRIMARY	PRIVILEGES	PROCEDURE
PURGE	READ	REAL	REFERENCES
REGEXP	RENAME	REPLACE	REQUIRE
RESTRICT	RETURNS	REVOKE	RIGHT
RLIKE	SELECT	SET	SHOW
SMALLINT	SONAME	SQL_BIG_RESULT	SQL_CALC_FOUND_ROWS
SQL_SMALL_RESULT	SSL	STARTING	STRAIGHT_JOIN
STRIPED	TABLE	TABLES	TERMINATED
THEN	TINYBLOB	TINYINT	TINYTEXT
TO	TRAILING	UNION	UNIQUE
UNLOCK	UNSIGNED	UPDATE	USAGE

USE	USER_RESOURCES	USING	VALUES
VARBINARY	VARCHAR	VARYING	WHEN
WHERE	WITH	WRITE	XOR
YEAR_MONTH	ZEROFILL		

Los siguientes símbolos (de la tabla anterior) no se permiten en ANSI SQL pero se permiten en MySQL como nombres te tablas y/o columnas. Ello es debido a que algunos de estos nombres son muy naturales y muchos usuarios los utilizan:

- ACTION
- BIT
- DATE
- ENUM
- NO
- TEXT
- TIME
- TIMESTAMP

6.2. Tipos de valores de las columnas

MySQL soporta diferentes tipos de columnas, que pueden ser clasificados en tres categorías: valores numéricos, valores de fecha y hora, y cadenas. En esta sección se echará un vistazo a los tipos existentes y luego se resumirá los requerimientos de almacenamiento para cada tipo de columna, posteriormente aporta descripción más detallada de las propiedades de los tipos en cada categoría. El vistazo es intencionadamente breve. Las descripciones más detalladas deben ser consultadas como información adicional sobre tipos de columnas particulares, tales como formatos permitidos en los cuales se puedan especificar valores.

Los tipos de columna permitidos por MySQL están listados más abajo. Las siguientes letras se utilizan en las descripciones:

M	Indica el máximo espacio de representación. El máximo tamaño legal es de 255.
D	Se aplica en tipos de coma flotante e indica el número de dígitos que siguen al punto decimal. El valor máximo posible es de 30, pero no debería ser superior a M-2.

Los corchetes (‘[‘ y ‘]’) indican que partes de las especificaciones son opcionales.

Comentar que si especificas ZEROFILL para una columna, MySQL añadirá automáticamente el atributo UNSIGNED a la columna.

Atención: Debes ser cauto al utilizar la sustracción entre valores enteros de diferentes columnas, donde una de ellas es de tipo UNSIGNED, ya que el resultado será también UNSIGNED! Puedes ver la sección 6.3.5 [Cast functions].

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.

BIT, BOOL

Son sinónimos para TINYINT(1).

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

Un entero pequeño. El rango con signo es de –32768 a 32767. El rango sin signo es de 0 a 65536.

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

Es un entero de tamaño medio. El rango con signo es de –8388608 a 83888607. El rango sin signo 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]

Sinónimo de INT.

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

Entero largo. El rango con signo es de –9223372036854775808 a 9223372036854775807. El rango sin signo es de 0 a 18446744073709551615.

Hay ciertas cosas sobre las que deberías tener cierta precaución sobre las columnas BIGINT:

- Todos los cálculos aritméticos están hechos utilizando valores BIGINT o DOUBLE, con lo que no deberías utilizar BIGINT sin signo [UNSIGNED] mayores de 9223372036854775807 (63 bits) excepto en funciones de bits. Si haces esto, algunos de los últimos dígitos del resultado pueden ser erróneos a causa del redondeo al convertir de BIGINT a DOUBLE.
    - MySQL 4.0 puede manejar los BIGINT en los siguientes casos:
        Utilizar enteros para almacenar valores grandes sin signo en una columna BIGINT.
        En MIN(big_int_column) y MAX(big_int_column).
        Al utilizar operadores (+,-+*,/, etc.) donde los dos operandos sean enteros.
- Siempre puedes almacenar un valor entero exacto en una columna BIGINT almacenándolo como cadena. En este caso, MySQL realizará una conversión de cadena a número que no implique una representación DOUBLE intermedia.
- Los operadores de suma, resta y multiplicación utilizarán la aritmética de BIGINT cuando los dos argumentos sean valores enteros. Esto implica que si multiplicas dos grandes enteros (o los resultados de funciones que retornan enteros) puedes recibir resultados inesperados cuando el resultado es superior a 9223372036854775807.

FLOAT(precisión) [UNSIGNED] [ZEROFILL]

Un valor de precisión de coma flotante puede tener 24 o menos decimales para un número de precisión simple, y entre 25 y 53 para un valor de precisión doble. Estos tipos son como los tipos FLOAT y DOUBLE descritos posteriormente. FLOAT(X) tiene el mismo rango que los correspondientes a FLOAT y DOUBLE, sólo que el tamaño de la representación y el número de decimales es indefinido.

En la versión 3.23 de MySQL, éste es un verdadero valor de coma flotante. En versiones anteriores de MySQL, FLOAT(precisión) tiene siempre 2 decimales.

Comentaremos que utilizando FLOAT puede darte algunos problemas inesperados como todos los cálculos en MySQL que se realizan con doble precisión. Puedes ver la sección A.5.6 [No matching rows].

Esta sintaxis se da por compatibilidad con ODBC.

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

Un valor de coma flotante pequeño (precisión simple). Los valores permitidos son de –3.402823466E+38 a –1.175494351E-38, 0 y de 1.175494351E-38 a 3.402823466E+38. Si se especifica un valor sin signo, los valores negativos no se permiten. El valor M se refiere al tamaño de presentación, y D es el número de decimales. FLOAT sin argumentos o FLOAT(x), donde x<=24 da un valor de coma flotante con precisión simple.

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

Valor de coma flotante de tamaño normal. Los valores permitidos son de:

–1.797693138623157E+308 a –2.2250738585072014E-308 y de
2.2250738585072014E-308 a 1.797693138623157E+308. Si se especifica la cláusula UNSIGNED,, los valores negativos no se permitirán. M es el tamaño de presentación de datos, y D es el número de decimales. Un valor DOUBLE sin argumentos o un FLOAT(X) donde el número de decimales oscila entre 25 y 53 da como resultado un valor de coma flotante de precisión doble.

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

Son sinónimos de DOUBLE.

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

Valor de coma flotante no empaquetado [?]. Funciona como una columna de tipo CHAR: “no empaquetado” significa que el número es almacenado como una cadena, utilizando un carácter para cada dígito del valor. El punto decimal y, para valores negativos, el signo ‘- ‘, no se contabilizan en M (aunque el espacio para ellos se reserva). Si D es 0, los valores no tendrán punto decimal o parte fraccionaria. El máximo rango para valores DECIMAL es el mismo que para DOUBLE, pero el rango actual para una columna DECIMAL dada puede ser restringido por la elección de M y D. Si se especifica un UNSIGNED, los valores negativos.

Si D se omite, el valor por defecto es 0. Si M se omite, el valor por defecto es 10. En las versiones anteriores a MySQL 3.23, el argumento M debe incluir el espacio necesario para el signo y el punto decimal.

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

Son sinónimos de DECIMAL.

DATE

Valor de fecha. El rango soportado es '100-01-01' a '9999-12-31'. MySQL representa el valor de la fecha en formato YYYY-MM-DD, pero te permite asignar valores a columnas de tipo DATE con cadenas o números. Puedes ver la seción 6.2.2.2 [DATETIME].
DATETIME

Una combinación de fecha y hora. El rango soportado es de '1000-01-01 00:00:00' a '9999-12-31 23:59:59'. MySQL representa los datos de tipo DATETIME en formato 'YYYY-MM-DD HH:MM:SS', pero te permite asignar valores a las columnas DATETIME utilizando tanto cadenas como números. Puedes ver la sección 6.2.2.2 [DATETIME].

TIMESTAMP[(M)]

Tipo Timestamp [?]. El rango es de '1970-01-01 00:00:00' hasta algún momento del año 2037. MySQL representa los valores TIMESTAMP en formatos YYYYMMDDHHMMSS, YYMMDDHHMMSS, YYYYMMDD, o YYMMDD, dependiendo de si M vale 14 (o no se indica), 12, 8 o 6, te permite asignar valores a columnas TIMESTAMP utilizando tanto cadenas o números. Una columna TIMESTAMP es útil para grabar datos de fecha y hora de una sentencia INSERT o UPDATE, ya que su valor se actualiza automáticamente a la hora y fecha de la última operación ejecutada, si no es que tú le das un valor. Puedes actualizarlo a la fecha y hora actual asignándole un valor NULL. Puedes ver la sección 6.2.2. [Date and time types].

El argumento M afecta sólo al modo en que la columna TIMESTAMP es representada; sus valores siempre se almacenan utilizando 4 bytes cada uno.

Cabe anotar que las columnas de TIMESTAMP(M) donde M es 8 o 14 se consideran números, mientras que otros valores se consideran cadenas. Esto se hace simplemente para asegurarse que puedas volcar fiablemente y recuperar estos datos de la tabla. Puedes ver la sección 6.2.2.2. [DATETIME].

TIME

Una hora. El rango es de '-838:59:59' a '838:59:59'. MySQL representa los valores TIME en formato HH:MM:SS, pero te permite asignarle valores utilizando cadenas o números. Puedes ver la sección 6.2.2.3.

YEAR[(2|4)]

Un año en formato de 2 o 4 dígitos. Los valores permitidos son de 1901 a 2155 en el formato de 4 dígitos, y de 1970 a 2069 en el formato de dos dígitos. MySQL representa los valores YEAR en formato YYYY, pero te permite asignar valores a YEAR utilizando tanto cadenas como números. (El tipo YEAR no está disponible en las versiones anteriores a 3.22 de MySQL). Puedes ver la sección 6.2.2. [YEAR]

[NATIONAL] CHAR(M) [BINARY]

Una cadena de longitud fija que siempre tiene espacios añadidos en la parte derecha para ajustarse a la longitud de almacenamiento especificada. El rango de M es de 0 a 255 caracteres (1 a 255 en versiones anteriores a 3.23 de MySQL). Los espacios precedentes son eliminados cuando el valor se recupera. los valores CHAR están ordenados y comparados con diferenciación de mayúsculas y minúsculas de acuerdo con el juego de caracteres por defecto, a no ser que se indique la palabra clave BINARY.

NATIONAL CHAR (o su forma equivalente, NCHAR) es la forma de ANSI SQL para definir una columna CHAR que debe utilizar el juego de caracteres CHARACTER por defecto. Este es el valor por defecto de MySQL.

CHAR es una abreviatura de CHARACTER.

MySQL te permite crear una columna de tipo CHAR(0). Esta es especialmente útil cuando tienes que presentar concordancia con ciertas aplicaciones antiguas que dependen de la existencia de una columna pero que no utilizan actualmente el valor. Esto es especialmente interesante cuando necesitas una columna donde sólo pueden tomar dos valores: Un CHAR(0), que no se define como NOT NULL, ocupará sólo un bit y puede tomar sólo 2 valores: NULL o " ". Puedes ver la sección 6.2.3.1 [CHAR].

CHAR Es un sinónimo de CHAR(1).

[NATIONAL] VARCHAR(M) [BINARY]

Una cadena de longitud variable. El rango de M es de 0 a 255 caracteres (1 a 255 en versiones anteriores a MySQL 4.0.2). Los valores VARCHAR son almacenados y comparados diferenciando mayúsculas y minúsculas a menos que indique la palabra clave BINARY. Puedes ver la sección 6.5.3.1 [Silent column changes].

Nota: los espacios precedentes se eliminan cuando el valor se almacena (Esto difiere de la especificación ANSI SQL).

VARCHAR es una abreviatura de CHARACTER VARYING. Puedes ver la sección 6.2.3.1. [CHAR].

TINYBLOB
TINYTEXT

Una columna de tipo BLOB o TEXT con una longitud máxima de 255 (2^8-1) caracteres. Puedes ver la sección 6.5.3.1. [Silent Column Changes] y la sección 6.2.3.2. [BLOB]

BLOB
TEXT

Una columna de tipo BLOB o TEXT con longitud de 65535 (2^{16}-1) caracteres. Puedes ver la sección 6.5.3.1. [Silent column changes], y la sección 6.2.3.2. [BLOB].

MEDIUMBLOB
MEDIUMTEXT

Una columna BLOB o TEXT con una longitud de 16777215 (2^{24}-1) caracteres. Puedes ver la sección 6.5.3.1. [Silent column changes] y 6.2.3.2. [BLOB].

LONGBLOB
LONGTEXT

Una columna BLOB o TEXT con longitud para 4294967295 (2^{32}-1) caracteres. Puedes ver la sección 6.5.3.1. [Silent Column changes]. Debido al protocolo de cliente/servidor y a las tablas

MyISAM tienen actualmente un límite de 16M por paquete de comunicación o fila de tabla, este tipo de dato no puede ser utilizado en toda su extensión. Puedes ver la sección 6.2.3.2. [BLOB]

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

Una enumeración. Un objeto de cadena que puede tener un solo valor, escogido de una lista de valores 'valor1, 'valor2',...,NULL o el valor especial "" de error. Una columna ENUM puede tener un máximo de 65535 valores diferentes. Puedes ver la sección 6.2.3.3 [ENUM].

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

Una lista. Un objeto de cadena que puede tener 0 o más valores, cada una de las cuales debe ser escogida de la lista de valores 'valor1', 'valor2', ... Una columna SET puede tener un máximo de 64 miembros. Puedes ver la sección 6.2.3.1. [SET].

6.2.1. Valores numéricos.

MySQL soporta todos los tipos numéricos de la ANSI/ISO SQL92. Estos tipos incluyen los tipos numéricos exactos (NUMERIC, DECIMAL, INTEGER, y SMALLINT), igualmente como tipos numéricos aproximados (FLOAT, REAL, y DOUBLE PRECISION). La palabra clave INT es un sinónimo de INTEGER, y la palabra clave DEC es un sinónimo para DECIMAL.

Los tipos NUMERIC y DECIMAL son implementados como el mismo tipo en MySQL, tal como permite el estándar SQL92. Se utilizan para los valores en los que es importante preservar la precisión exacta, por ejemplo en información monetaria. Cuando se declara una columna con unos de estos datos, la precisión y la escala pueden ser (y habitualmente lo son) especificada; por ejemplo:

salary DECIMAL(5,2)

En este ejemplo, 5 (precisión) representa el número de digitos decimales significativos que van a ser almacenados como valores, y 2 (escala) indica el número de dígitos que se almacenarán después del punto decimal. En este caso, el rango de valores que pueden ser almacenados en la columna salary es de –99.99 a 99.99. (MySQL puede almacenar actualmente valores por encima de 999.99 en esta columna debido a que no necesita almacenar el signo para números positivos).

Con ANSI/ISO SQL92, la sintaxis DECIMAL(p) es equivalente a DECIMAL(p,0). De manera similar, la sintaxis DECIMAL es equivalente DECIMAL(p,0), donde la implementación es permitida para decidir el valor de p. MySQL no soporta actualmente estas variantes de las formas de los tipos DECIMAL/NUMERIC. Esto no es, por regla general, un problema serio, ya que los beneficios principales de estos tipos derivan de la habilidad de contralar tanto la precisión como la escala explícitamente.

Los valores de DECIMAL y NUMERIC son almacenadas como cadenas, no como los números binarios de coma flotante, con la intención de preservar la precisión decimal de estos valores. Cuando una columna se asigna a un valor con más dígitos posteriores al punto decimal que son permitidos por la escala especificada, el valor es redondeado a esta escala. Cuando a las columnas con valores DECIMAL o NUMERIC se asigna al valor la magnitud del cual excede el rango implicado por la precisión y escala especificada (o por defecto), MySQL almacena el valor representando el punto final correspondiente al rango.

Como a extensión del estándar ANSI/ISO SQL92, MySQL también soporta los tipos enteros TINYINT, MEDIUMINT y BIGINT como se listaba en las tablas anteriores. Otra extensión soportada por MySQL por opcionalidad especificando la longitud de la representación de un valor entero entre paréntesis siguiendo a la plabra clave del tipo (por ejemplo INT(4)). Esta especificación opcional del tamaño es usuada para ajustar por la izquierda la representación de valores cuyo tamaño es inferior que el especificado en la columna, pero no restringe el rango de los valores que pueden ser almacenados en la columna, ni el número de dígitos que serán representados por valores en los que su tamaño excede el especificado para la columna. Cuando se utiliza en conjunción con el atributo de extensión opcional ZEROFILL, un valor como 4 es recuperado como 0004. Cabe anotar que si almacenas valores más largos que el ancho de representación especificado en una columna integer, puedes experimentar problemas cuando MySQL genere datos temporales para ciertas acciones de [join], ya que en estos casos MySQL cree que los datos se ajustaron al ancho de columna original.

Todos los tipos enteros pueden tener un atributo opcional (no estándar), UNSIGNED. Valores UNSIGNED pueden ser utilizados cuando quieres permitir sólo valores positivos en una columna y necesitas un rango de valor numérico algo mayor para la columna.

Como en MySQL 4.0.2., los tipos de coma flotante pueden ser también unsigned. Como sucede con los tipos enteros, este atributo previene que los valores negativos sean almacenados. A diferencia de los tipos enteros, el valor superior de los valores de columna se mantiene igual.

El tipo FLOAT es utilizado para representar tipos aproximados de valores numéricos. El estándar ANSI/ISO SQL92 permite una especificación opcional de la precisión (pero no del rango del exponente) en bits, indicados entre paréntesis después de la palabra clave FLOAT. La implementación de MySQL también soporta esta especificación opcional de precisión. Cuando la palabra clave FLOAT es utilizada como tipo de valor de una columna sin indicar una precisión, MySQL utiliza 4 bytes para almacenar los valores. Una variante de sintaxis también soportada, es la que indica dos valores entre paréntesis posteriores a la palabra FLOAT. Con esta opción, el primer número continua representando los requerimientos de almacenamiento en bytes para el valor, y el segundo número especifica el número de dígitos a almacenar y representados que siguen al punto decimal (como en los casos de DECIMAL y NUMERIC). Cuando a MySQL se se consulta para almacenar un número por una columna con más dígitos decimales que siguen al punto decimal de los especificados para la columna, el valor se redondea para eliminar los dígitos de más al almacenar el dato.

Los tipos de datos REAL y DOUBLE PRECISION no aceptan especificaciones de precisión. Como una extensión del estándar ANSI/ISO SQL92, MySQL reconoce DOUBLE como un sinónimo para el tipo DOUBLE PRECISION. En contraste con el requerimiento del estándar, según el cual la precisión para REAL debe ser menor que para DOUBLE PRECISION, MySQL implementa ambos como tipos de coma flotante con doble precisión de 8 bytes (al funcionar en "ANSI mode"). Para conseguir la máxima portabilidad, el código que se requiera para el almacenamiento de datos aproximados debería utilizar los tipos FLOAT o DOUBLE PRECISION sin especificación de precisión o número de valores decimales.

Al ser consultado para almacenar un valor en una columna numérica que está fuera del rango permitido en el tipo de tal columna, MySQL recorta el valor al punto máximo del rango y almacena el resultado en substitución del primero.

Por ejemplo, el rango de una columna INT es –2147483648 a 2147483647. Si tratas de insertar –9999999999 en una columna INT, el valor es al límite negativo del rango, y se almacena –2147483648. De un modo similar, si tratas de insertar 9999999999, se almacena 2147483647.

Si la columna INT no tiene signo [UNSIGNED], el tamaño del rango de la columna es el mismo pero sus límites máximos y mínimos son 0 y 4294967295. Si tratas de almacenar –9999999999 y 9999999999, los valores almacenados serán 0 y 4294967296 (?).

Las conversiones que ocurren al recortar se expresan como alertas [warnings] para las sentencias ALTER TABLE, LOAD DATA INFILE, UPDATE y INSERT de varias filas.

6.2.2. Tipos de fecha y hora.

Los tipos de fecha y hora son DATETIME, DATE, TIMESTAMP YEAR. Cada uno de ellos tiene un rango de valores legales, así como un valor "cero" que se utiliza al especficar un valor ilegal. MySQL te permite almacenar ciertas fechas que no son estrictamente correctas, como por ejemplo 1999-11-31. La razón es que pensamos que es responsabilidad de la aplicación manejar el control de la fecha, y no de los servidores de MySQL. Para agilizar el chequeo de las fechas, MySQL sólo chequea que el valor del mes oscile entre 0 y 12, y que el rango del día esté entre 0 y 31. Los anteriores rangos son definidos así debido a que MySQL te permite almacenar, en una columna de tipo DATE o DATETIME, fechas en las que el día o el mes puede ser 0. Esta posibilidad se hace extremamente útil para las aplicaciones en las que existe la necesidad de almacenar fechas como 1999-00-00 o 1999-01-00. (No puedes esperar el retorno de valores correctos en funciones como DATE_SUB() o DATE_ADD para fechas como estas).

Aquí hay algunas consideraciones generales a tener en cuenta en el momento de trabajar con datos de fecha y hora:

- MySQL retorna valores para un tipo de fecha y hora determinado en un formato estándar, pero trata de interpretar una cierta variedad de formatos de valores que le aportes (por ejemplo, como cuando especificas un valor que debe ser asignado o comparado a un tipo fecha/hora). Aún así, sólo los formatos especificados en las próximas secciones son soportados. Se espera de ti que proporciones valores legales, y que los resultados impredecibles ocurran en el caso que utilices fechas con otros formatos.
- A pesar de que MySQL trata de interpretar valores en varios formatos, siempre espera que el año sea el valor más a la izquierda. Las fechas han de ser proporcionadas en el orden año-mes-día (por ejemplo 98-09-04), en vez de mes-día-año o día-mes-año, utilizados comúnmente en otros casos (ejemplos: '09-04-98', '04-09-98').
- MySQL convierte automáticamente un tipo de fecha y hora a un número si el valor es utilizado en un contexto numérico, y vice-versa.
- Cuando MySQL encuentra un valor para un tipo de fecha y hora que se encuentra fuera del rango o en cualquier caso que su valor es ilegal para el tipo (puedes ver el inicio de esta sección), convierte el valor al "cero" correspondiente al tipo. (La excepción es que los valores de tipo TIME que están fuera de rango se ajustan al límite máximo del rango de TIME). La tabla siguiente presenta el formato del valor "cero" de cada tipo:

Tipo de columna	**Valor "cero"**
DATETIME	' 0000-00-00 00:00:00'
DATE	' 0000-00-00'
TIMESTAMP	00000000000000 (la longitud depende del tamaño de representación)
TIME	' 00:00:00'
YEAR	0000

- El valor "cero" es especial, pero puedes almacenar o referirte a ellos explícitamente utilizando los valores presentados en la tabla. También puedes hacerlo utilizando los valores '0' o 0, que son más sencillos de escribir.
- Los valores de fecha u hora "cero" utilizados a través de MyODBC son convertidos automáticamente a NULL en las versiones 2.50.12 y superiores de MyODBC, ya que ODBC no puede manejar tales tablas.

6.2.2.1. Aspectos y valores de fecha para el efecto 2000

MySQL en sí misma está a salvo del efecto 2000 (Puedes ver la sección 1.2.5. [Year 2000 compliance]), pero los valores de introducción presentados a MySQL pueden no serlo. Cualquier introducción que contenga un año representado con dos dígitos es ambigua, ya que se desconoce su siglo. Tales valores deben ser interpretados en un formato de 4 dígitos ya que MySQL almacena internamente los años utilizando 4 dígitos.

Para los tipos DATETIME, DATE, TIMESTAMP, y YEAR, MySQL interpreta las fechas con valores anuales ambiguos utilizando las siguientes reglas:

- Valores anuales entre 00-69 se convierten a 2000-2069
- Valores anuales de 71-99 se convierten a 1971-1999.

Recuerda que estas reglas proporcionan criterios razonables sobre lo que significan tus datos. Si la heurística utilizada por MySQL no produce valores correctos, deberías proveer entradas sin ambigüidades que contuvieran valores anuales de 4 dígitos.

ORDER BY ordenará los dígitos anuales de los tipos de YEAR/DATE/DATETIME con propiedad a el criterio anterior.

Hay que tener en cuenta que algunas funciones como MIN() y MAX() convertirán TIMESTAMP/DATE a un valor numérico. Esto significa que un valor TIMESTAMP con un año de dos dígitos no funcionará correctamente con estas funciones. El ajuste en este caso se basa en convertir los datos TIMESTAMP/DATE a un formato de año de 4 dígitos o utilizar algo como MIN(DATE_ADD(timestamp,INTERVAL 0 DAYS)).

6.2.2.2. Los tipos DATETIME,DATE y TIMESTAMP

Los tipos DATETIME, TIME y TIMESTAMP se han explicado. Esta sección describe sus características, en qué se asemejan y en qué difieren.

El tipo DATETIME se utiliza cuando necesitas valores que contienen información de fecha y hora a la vez. MySQL recupera y presenta los tipos DATETIME en formato 'YYYY-MM-DD HH:MM:SS'. El rango soportado es '1000-01-01 00:00:00' a '9999-12-31 23:59:59' ("Soportado" significa que a pesar de que en versiones anteriores los valores podrían funcionar, esto no es una garantía de que ello suceda).

El tipo DATE es utilizado cuando sólo necesitas un valor de fecha, sin la parte horaria. MySQL recupera y representa los valores de DATE en formato 'YYYY-MM-DD'. El rango soportado '1000-01-01' to '9999-12-31'.

La columna de tipo TIMESTAMP proporciona un valor que es aplicable automáticamente en operaciones como INSERT y UPDATE con la fecha y hora actuales. Si tienes múltiples columnas TIMESTAMP, sólo la primera es actualizada automáticamente.

La actualización automática de la primera columna TIMESTAMP tiene lugar bajo cualquiera de las siguientes condiciones:

- La columna no se especifica explícitamente en una sentencia INSERT o LOAD DATA INFILE.
- La columna no se especifica en una sentencia UPDATE y en otras cambia el valor. (Cabe decir que una sentencia UPDATE que indique que el valor de la columna sigue el mismo no cambia el valor, ya que MySQL ignora la actualización por una cuestión de eficiencia).
- Si explícitamente das el valor NULL a la columna TIMESTAMP.

El resto de columnas TIMESTAMP que no sean la primera pueden ser situadas a la fecha y hora actuales, asignándoles cualquiera de los dos valores siguientes: NULL o NOW().

Puedes asignar un valor diferente a la fecha actual en una columna simplemente asignándoselo explícitamente. Esto es un evento para la primera columna. Puedes utilizar esta propiedad si, por ejemplo, quieres que la columna TIMESTAMP guarde la fecha y hora en la que se ha actualizado una fila, pero que no sea cambiado en ninguna actualización más:

- Deja que sea MySQL la que le asigne un valor en el momento de crear la fila. El valor asignado será el de la fecha y hora actuales.
- Cuando realices las siguientes actualizaciones sobre esta fila, asigna el valor explícitamente a la TIMESTAMP con el valor que ya tiene.

Por otro lado, puedes considerar esto tan sencillo como usar una columna DATETIME inicializada con NOW() cuando la fila es creada y dejarla así en las siguientes actualizaciones.

Los valores TIMESTAMP pueden oscilar del inicio de 1970 hasta en algún momento del año 2037, con un nivel de resolución de un segundo. Los valores son representados como números.

El formato en el que MySQL recupera y representa los valores de TIMESTAMP depende del tamaño de representación, como se ilustra en la tabla siguiente. El formato TIMESTAMP "entero" es de 14 dígitos, pero las columnas TIMESTAMP pueden ser creadas con tamaños de representación más cortos:

Tipo de columna	Formato de visualización
TIMESTAMP(14)	YYYYMMDDHHMMSS
TIMESTAMP(12)	YYMMDDHHMMSS
TIMESTAMP(10)	YYMMDDHHMM
TIMESTAMP(8)	YYYYMMDD
TIMESTAMP(6)	YYMMDD
TIMESTAMP(4)	YYMM
TIMESTAMP(2)	YY

Todas las columnas TIMESTAMP tienen el mismo tamaño de almacenamiento, independientemente del tamaño de visualización. Los tamaños más habituales son 6, 8, 12 y 14. Puedes especificar tamaños arbitrarios en el momento de la creación de la tabla, pero los valores de 0 mayores de 14 se reubican en 14. Los tamaños con valores impares en el rango de 1 a 12 se ajustan a su valor par superior.

Puedes especificar valores DATETIME, DATE y TIMESTAMP utilizando cualquiera de los siguientes formatos comunes:

- Como cadena, tanto en el formato 'YYYY-MM-DD HH:MM:SS' como de 'YY-MM-DD HH:MM:SS'. Una sintaxis más "relajada" permitida, ya que se permite cualquier signo de puntuación como delimitador de cada una de las partes de la fecha y la hora. Por ejemplo, '98-12-31 11:30:45', '98.12.31 11+30+45', '98/12/31 11*30*45', y '98@12@31 11^30^45' son equivalentes.
- Como cadena, tanto en el formato 'YYYY-MM-DD' como en 'YY-MM-DD'. La sintaxis "relajada" se permite también en este caso: así, '98-12-31', '98.12.31', '98/12/31', y '98@12@31' son equivalentes.
- Como cadena sin delimitadores, tanto en el formato 'YYYYMMDDHHMMSS' com en 'YYMMDDHHMMSS' dando por hecho que la cadena tiene sentido como fecha. Por ejemplo, '19970523091528' y '970523091528' son interpretadas como '1997-05-23 09:15:28', pero '971122129015' es un valor ilegal (la parte de los minutos no tiene sentido) y se procesa como '0000-00-00 00:00:00'.
- Como cadena sin delimitadores, tanto en el formato 'YYYYMMDD' como en 'YYMMDD', suponiendo que la cadena tiene sentido como fecha. Por ejemplo, '19970523' y '970523' se consideran '1997-05-23', pero '971332' es ilegal (ni el valor del mes ni el del día tienen sentido) y se procesa como '0000-00-00'.
- Como número, tanto en formato YYYYMMDDHHMMSS com en YYMMDDHHMMSS, suponiendo que el número tenga sentido como fecha. Por ejemplo, 19830905132800 y 830905132800 se interpretan como '1983-09-05 13:28:00'.
- Como número, tanto en el formato YYYYMMDD como en YYMMDD, suponiendo que el número tiene sentido como fecha. Por ejemplo, 19830905 y 830905 se interpretan como '1983-09-05'.
- Como resultado de una función que retorna un valor aceptable en un contexto de DATE, DATETIME o TIMESTAMP, com NOW() o CURRENT_DATE.

Los valores ilegales de DATETIME, TIME o TIMESTAMP se convierten al valor "cero" de cada tipo.

Para valores especificados como cadenas que incluyen delimitadores de las partes de fecha, no es necesario especificar dos dígitos para los valores de día o mes inferiores a 10. '1979-6-9' es lo mismo que '1979-06-09'. De un modo similar, los valores especificados como cadenas que incluyen delimitadores entre las partes de la hora, no es necesario especificar dos dígitos por hora, minuto o segundo, cuando los valores son inferiores a 10. '1979-10-30 1:2:3' es lo mismo que '1979-10-30 01:02:03'.

Los valores especificados como números deberían tener una longitudo de 6, 8, 12 o 14 dígitos. Si se le da una longitud de 8 o 14 dígitos, se asume que el formato es del tipo YYYYMMDD o YYYYMMDDHHMMSS, y que el año viene dado por los primeros 4 dígitos. Si el número de dígitos es de 6 o 12, se asume que el formato es YYMMDD o YYMMDDHHMMSS y que el año viene dado por sus dos primeros dígitos. Los números que tienen una de estas longitudes se interpretan como si se rellenaran de ceros en las posiciones más significativas para ajustarse a la longitud más parecida.

Los valores especificados sin delimitadores se interpretan utilizando su longitud dada. Si la cadena tiene 8 o 14 caracteres, se asume que los primeros 4 dígitos indican el año. En el resto de casos, el año viene indicado por los dos primeros caracteres. La cadena se interpreta de izquierda a derecha para encontrar los valores del año, mes, día, hora, minutos y segundos, con tantas partes como se representen en la cadena. Esto implica que do deberías usar cadenas que tengan menos de 6 caracteres. Por ejemplo, si especificas '9903', pensando

que el valor representa al mes de marzo del año 1999, te encontrarás con que MySQL inserta un "cero" en tu tabla. Ello es debido a que los valores de año y mes son 99 y 3, pero que no se halla la parte que indica el día, con lo que el valor, como fecha, no es legal.
Las columnas TIMESTAMP almacenan valores legales toda la precisión con la que el el valor fue especificado, independientemente del tamaño de visualización. Ello tiene varias implicaciones:

- Especifica siempre el año, mes y día, incluso si los tipos de columna especificados son TIMESTAMP(4) o TIMESTAMP(2). En cualquier otro caso, el valor como fecha no se considerará legal, y se guardará un "0".
- Si utilizas ALTER TABLE para ampliar o estrechar una columna TIMESTAMP, la información "oculta" podrá ser visualizada.
- De un modo similar, estrechar una columna TIMESTAMP no causa una pérdida de información, excepto en el sentido que se visualiza menos información al mostrar los valores.
- A pesar de que los valores TIMESTAMP son almacenados en precisión completa, la única función que opera directamente con el valor subyacente es UNIX_TIMESTAMP(). Otras funciones operan con el valor recuperado y formateado. Ello implica que no puedes utilizar funciones tales como HOUR() o SECOND() a menos que la parte relevante del valor de TIMESTAMP se incluya en el valor formateado. Por ejemplo, la parte HH de TIMESTAMP no se visualiza si el tamaño de visualización no es de 10 al menos, luego tratar de utilizar HOUR() en valores TIMESTAMP más cortos produce un resultado sin sentido.

Puedes extender asignaciones de algunos tipos de fecha y hora diferente. Sin embargo, pueden tener lugar algunas alteraciones del valor, o pérdida de información:

- Si asignas un valor DATE a un objeto DATETIME o TIMESTAMP, la parte del tiempo resultante del valor se sitúa en '00:00:00', debido a que el valor DATE no contiene información sobre la hora.
- Si asignas un valor DATETIME o TIMESTAMP a un objeto DATE, la parte de la hora del valor resultante se borra, ya que el tipo DATE no almacena información sobre la hora.
- Recuerda que a pesar de que los valores DATETIME, DATE y TIMESTAMP pueden ser especificados utilizando el mismo sistema de formatos, los tipos no tienen el mismo rango de datos. Por ejemplo, los valores TIMESTAMP no pueden ser anteriore a 1970 o posteriores al 2037. Esto significa que una fecha como '1968-01-01', mientras que es legal en tipos DATETIME y DATE, no lo es para los tipos TIMESTAMP, y en este caso se convertirá en un "0" si se asigna a un objeto.

Sé cauto con ciertas trampas al especificar valores de fecha:

- El formato relajado para especificar valores en cadenas puede ser decepcionante. Por ejemplo, un valor tal como '10:11:12' podría parecer un valor de hora debido al delimitador ':', pero si se utiliza en un contexto de fecha, será interpretado como '2010-11-12'. El valor '10:45:15' será convertido a '0000-00-00', ya que 45 no es un número válido de mes.
- El servidor MySQL sólo realiza el chequeo básico sobre la validez de una fecha: días del 0 al 31, meses del 0 al 12, años del 1000 a 1999. Cualquier fecha que no esté dentro de este rango, revertirá en 0000-00-00. Cabe anotar que ello te permite almacenar fechas desde un formulario sin un chequeo posterior. Para asegurarte de que una fecha es correcta, realiza el control en tu aplicación.

- Los valores de los años especificados con dos dígitos son ambiguos, debido a que se desconoce su siglo. MySQL interpreta los valores anuales de 2 dígitos utilizando las siguientes reglas:
    - Los valores anuales del rango 00-69 se convierten en 2000-2069.
    - Los valores anuales del rango 70-99 se convierten en 1970-1999.

6.2.2.3. El tipo TIME

MySQL recupera y visualiza los valores TIME en el formato 'HH:MM:SS' (o 'HHH:MM:SS' para valores altos de horas). Los valores de horas están en el rango de '- 838:59:59' a '838:59:59'. La razón por la que la parte de las horas puede ser tan larga es que el tipo TIME puede ser utilizado no sólo para representar la hora de un día (que debe ser inferior a 24 horas). Sino para el tiempo transcurrido o el intervalo de tiempo entre dos sucesos (que pueden ser muy superiores a 24 horas, en incluso negativos).

Puedes especificar valores de tiempos en una variedad de formatos:

- Como cadena en el formato 'D HH:MM:SS.fraction'. (Nota: MySQL no almacena la fracción en una columna de tipo TIME). Puedes utilizar también una de las siguientes sintaxis "relajadas": HH:MM:SS.fraction, HH:MM:SS, HH:MM, D HH:MM:SS, D HH:MM, D HH o SS. Aquí D es el valor en días entre 0 y 33.
- Como cadena sin delimitadores, en formato 'HHMMSS' suponiendo que como hora el valor tiene sentido. Por ejemplo, '101112' se entiende como '10:11:12'. Los siguientes formatos alternativos también son interpretados correctamente: SS, MMSS, HHMMSS, HHMMSS.fraction. MySQL aún no almacena las fracciones de segundo.
- Como resultado de una función que retorna un valor aceptable en el contexto de una variable TIME, como CURRENT_TIME.

Para valores de tiempo especificados como cadenas que incluyen delimitadores de las partes de la hora, no es necesario especificar dos dígitos para los valores de las horas. minutos o segundos inferiores a 10. '8:3:3' es equivalente a '08:03:02'.

Sé cauto al asignar valores TIME cortos a una columna TIME. Sin los dos puntos, MySQL interpreta los valores asumiendo que el valor más a la derecha representa los segundos. (MySQL interpreta los valores TIME como valores de tiempo transcurrido, más que como hora del día). Por ejemplo, podrías considerar que tanto '1112' como 1112 significan '11:12:00', pero MySQL interpreta '00:11:12'. De un modo similar, '12' y 12 se interpretan como '00:00:12'. Los valores de tiempo con dos puntos, en contraste, son tratados siempre como horas del día. Es decir, '11:12' se considera '11:12:00', y no '00:11:12'.

Los valores que se encuentren fuera del rango del tipo TIME pero que en cualquier caso son legales se ajustan a los valores límitos del rango. Por ejemplo, '- 850:00:00' y '850:00:00' se convierten a '- 839:59:59' y '838:59:59'.

Los valores ilegales de TIME se convierten a '00:00:00'. Nota que debido a que '00:00:00' es por sí mismo un valor legal para TIME, no es posible averiguar si el valor introducido ya era inicialmente 00:00:00, o si el valor que se quería introducir era ilegal [y por ello se introdujo el "cero"].

6.2.2.4. El tipo YEAR

Es un tipo de un byte de longitud utilizado para representar loas años.

MySQL recupera y visualiza los valores YEAR en formato YYYY. El rango es de 1901 a 2155. Puedes especificar valores YEAR con la siguiente variedad de formatos:

- Como cadena de 4 dígitos en el rango de '1901' a '2155'.
- Como valor de 4 dígitos en el rango de 1901 a 2155.
- Como cadena de dos dígitos, en el rango '00' a '99'. Los valores en los rangos '00' a '69' y de '70' a '99' se convierten a valores YEAR en los rangos 2000 a 2069 y de 1970 a 1999.
- Como número de dos dígitos, en el rango de 1 a 99. Los valores en el rango de 1 a 69 y 70 a 99 se convierten a los valores YEAR de los rangos 2001 a 2069, y 1970 a 1999, respectivamente. Se puede ver que el rango para dos dígitos es ligeramente diferente que para el rango de dos dígitos de las cadenas, debido a que no puedes especificar un cero directamente como un número, y debe ser interpretado como 2000. Debes especificar como cadena '0' o '00', o será interpretado como 0000.
- Como resultado de la función que retorne un valor aceptable en un contexto YEAR, como NOW().

Los valores ilegales de YEAR se convierten a 0000.

6.2.3. Tipos datos para cadenas

Los tipos de cadenas son CHAR, VARCHAR, BLOB, TEXT, ENUM y SET. Esta sección describe cómo trabajan estos tipos de datos, sus requerimientos de almacenamiento, y cómo utilizarlos en tus consultas.

6.2.3.1. Los tipos CHAR y VARCHAR

Los tipos CHAR y VARCHAR son similares, pero difieren en la forma en la que son almacenados y recuperados. La longitud de una columna CHAR se fija a la longitud que declares al crear la tabla. La longitud puede ser cualquier valor entre 1 y 255. (En la versión MySQL 3.23, la longitud del dato CHAR podía ser de 0 a 255). Cuando los valores CHAR son almacenados, son rellenados por la derecha con espacios en blanco hasta llegar a la longitud especificada. Cuando tales valores son recuperados, los espacios precedentes son eliminados. Los valores de las columnas VARCHAR son cadenas de longitud variable. Puedes declarar una columna VARCHAR para tener cualquier longitud entre 1 y 255, como en el caso de CHAR. Sin embargo, en contraste con CHAR, los valores VARCHAR son almacenados utilizando sólo los caracteres necesarios, más un byte que indica la longitud. Los valores no se rellenan; a cambio, los espacios precedentes son eliminados al guardar los datos. (Esta eliminación de espacios difiere de la especificación ANSI SQL).

Si asignas un valor a una columna CHAR o VARCHAR que exceda del máximo permitido, el valor se trunca hasta ajustarse.

La tabla siguiente ilustra las diferencias entre dos tipos de columnas visualizando el resultado de almacenar varios valores de cadenas en columnas con CHAR(4) y VARCHAR(4):

Valor	CHAR(4)	Almacenamiento requerido	VARCHAR(4)	almacenamiento requerido
''	' '	4 bytes	''	1 byte
' ab'	' ab '	4 bytes	' ab'	3 bytes

‘ abcd’	‘ abcd’	4 bytes	‘ abcd’	5 bytes
‘ abcdefgh’	‘ abcd’	4 bytes	‘ abcd’	5 bytes

Los valores recuperados desde las columnas con tipos CHR(4) y VARCHAR(4) serán los mismos en cada caso, porque los espacios precedentes se eliminan del valor CHAR en el momento de recuperar el dato.

Las columnas con valores CHAR y VARCHAR son ordenadas y comparadas diferenciando entre mayúsculas y minúsculas, a menos que el atributo BINARY fuera especificado en el momento de la creación de la tabla. El atributo BINARY significa que los valores de la columna son ordenados y comparados diferenciando entre mayúsculas y minúsculas pero aplicando el orden del código ASCII de la máquina en la que corre MySQL. BINARY no afecta al almacenamiento o recuperación de la información.

El atributo BINARY es expansivo. Esto implica que si una columna marcada como BINARY es utilizada en una expresión, el total de la expresión es comparada como una expresión BINARY.

MySQL puede cambiar silenciosamente el tipo de columna de CHAR a VARCHAR en el momento de la creación de la tabla. Puedes ver la sección 6.5.3.1 [Silent column changes].

6.2.3.2. Los tipos BLOB y TEXT

Un BLOB es un objeto binario extenso [Binary Large Object] que puede retener un volumen variable de datos. Los cuatro tipos BLOB, TINYBLOB, BLOB, MEDIUMBLOB, y LONGBLOB difieren sencillamente en el máximo tamaño de los valores que pueden almacenar.

Los cuatro tipos de TEXT TINYTEXT, TEXT, MEDIUMTEXT y LONGTEXT se corresponden con los cuatro tipos BLOB y tienen las mismas longitudes máximas y requerimientos de almacenaje. La única diferencia entre BLOB y TEXT es que la ordenación y comparación se realiza diferenciando mayúsculas y minúsculas en el caso de los valores BLOB, y sin diferenciar en los casos de TEXT. En otras palabras, TEXT es un BLOB en el que no se diferencian el mayúsculas y minúsculas.

Si asignas un valor a una columna BLOB o TEXT que excede el máximo especificado para el tipo de valor, el valor se trunca para ajustarse al máximo.

En la mayoría de casos, podemos entender el tipo TEXT como un tipo VARCHAR en el que su longitud es tan grande como se quiera. De un modo similar, podemos entender el tipo BLOB como un VARCHAR BINARY con menos restricciones de espacio. Las diferencias son que:

- Puedes tener índices en campos BLOB y TEXT con la versión MySQL 3.23.2 y posteriores. Las versiones más antiguas de MySQL no lo soportan.
- No se borran los espacios en blanco precedentes en los tipos BLOB y TEXT al almacenar los datos, como se hace en los campos VARCHAR.
- Las columnas BLOB y TEXT no pueden tener valores por defecto.

MyODBC define los valores BLOB como LONGVARBINARY y los valores TEXT como LONGVARCHAR.

Debido a que los valores BLOB y TEXT pueden ser extremamente largos, puedes acarrear con ciertas restricciones al utilizarlos:

- Si quieres utilizar las cláusulas GROUP BY o ORDER BY en campos TEXT y BLOB, debes convertir el valor de la columna en un objeto de longitud fijada. El sistema estándar de hacerlo es con la función SUBSTRING. Por ejemplo:

```
mysql> SELECT comment FROM tbl_name, SUBSTRING(comment,20) AS substr
            ORDER BY substr;
```

Si no lo haces, sólo los primeros max_sort_length bytes de la columna serán utilizados al ordenar. El valor por defecto de max_sort_length es de 1024; este valor puede ser modificado usando la opción –0 al iniciar el servidor de mysqld. Puedes agrupar en una expresión que agrupe valores BLOB y TEXT especificando la posición de la columna o utilizando un alias:

```
mysql> SELECT id,SUBSTRING(blob_col,1,100) FROM tbl_name GROUP BY 2;
mysql> SELECT id,SUBSTRING(blob_col,1,100) AS b FROM tbl_name GROUP BY b;
```

- El tamaño máximo de un objeto BLOB o TEXT es determinado por tal tipo, pero el valor más grande que puedes transmitir entre cliente y servidor se determina por el volumen de memoria disponible y el tamaño de los buffers de comunicaciones. Puedes cambiar el tamaño del buffer del mensaje, pero debes hacerlo tanto en el lado de cliente como en el de servidor.

Cabe anotar que cada valor BLOB o TEXT se representa internamente por un objeto alojado separadamente. Esto contrasta con el resto de tipos de columnas, para los que el almacenamiento tiene lugar una vez por columna al abrir la tabla. [?]

6.2.3.3. El tipo ENUM

ENUM es un objeto de cadena cuyo valor normalmente es escogido entre una lista de valores permitidos que son enumerados explícitamente en la especificación de la columna en el momento de la creación de la tabla.

El valor puede ser también la cadena vacía (“”) o NULL bajo ciertas circunstancias:

- Si insertas un valor inválido dentro de un ENUM (esto es, una cadena no presente en la lista de valores permitidos), se inserta una cadena vacía como un valor especial de error. Esta cadena puede distinguirse de una cadena vacía “normal” por el hecho que la cadena tiene el valor numérico 0. Trataremos este tema posteriormente.
- Si se declara un ENUM como NULL, NULL es un valor legal también para la columna, y el valor por defecto es NULL. Si se declara un ENUM como NOT NULL, el valor por defecto es el primer elemento de la lista de valores permitidos.

Cada valor de la enumeración tiene un índice:

- Valores de la lista de elementos permitidos en la especificación de la columna se numeran a partir del 1.
- El valor del índice de una cadena vacía de error es 0. Ello implica que puedes usar la siguiente sentencia SELECT para encontrar las filas en las cuales se han asignado valores ENUM inválidos:

  ```
  mysql> SELECT * FROM tbl_name WHERE enum_col=0;
  ```

- El índice del valor NULL es NULL.

Por ejemplo, una columna especificada como ENUM("uno","dos","tres") puede tener cualquiera de los valores presentados aquí. Se indica el índice de cada valor:

Valor	índice
NULL	NULL
""	0
" uno"	1
" dos"	2
" tres"	3

Una enumeración puede tener un máximo de 65535 elementos.

A partir de la versión 3.23.51 los espacios finales son borrados de los valores ENUM cuando se crea la tabla.

El uso de mayúsculas y minúsculas es irrelevante cuando asignas valores a una columna ENUM. Sin embargo, los valores recuperados de la columna posteriormente tienen la misma combinación de mayúsculas y minúsculas que los especificados para los valores válidos indicados durante la creación de la tabla.

Si recuperas un ENUM en un contexto numérico, se retorna el índice del valor del campo. Por ejemplo puedes recuperar valores numéricos de una columna ENUM como esta:

mysql> SELECT enum_col+0 FROM tbl_name;

Si almacenas un número en un ENUM, el número es tratado como índice, y el valor almacenado es el miembro de la enumeración con tal índice. (Sin embargo, esto no funcionará con LOAD DATA, que trata todas las entradas como cadenas). No es recomendable almacenar números en un campo ENUM, porque ello confundiría las cosas.

Los valores ENUM se ordenan de acuerdo con el orden en el que los miembros de la enumeración se listaron en la especificación de la columna. (En otras palabras, los valores ENUM se ordenan de acuerdo con sus índices). Por ejemplo, "a" se ordena antes de "b" para el ENUM("a", "b"), pero "b" se ordena antes que "a" en ENUM("b", "a"). La cadena vacía se ordena antes que las cadenas no vacías, y los valores NULL se ordenan antes que todos los valores de la enumeración.

Si quieres tener todos los valores posibles de una columna ENUM, deberías usar: SHOW COLUMNS FROM tbl_name LIKE enum_column_name y interpretar la definición de ENUM en la segunda columna.

6.2.3.4. El tipo SET

Es un objeto de cadena que puede tener cero o más valores, cada uno de los cuales debe ser escogido de una lista de valores permitidos, especificados cuando se creó la tabla. Los valores de las columnas SET que consisten en múltiples miembros son especificados con los valores separados por comas (","). Una consecuencia de ello es que los valores de los miembros del SET no pueden contener comas en sí mismos.

Por ejemplo, una columna especificada como SET("uno", "dos") NOT NULL puede tener cualquiera de los siguientes valores:

""

“ uno”
“ dos”
“ uno,dos”

Un SET puede tener un máximo de 64 miembros.

A partir de la versión 3.23.51 los espacios sobrantes se borran automáticamente de los valores SET cuando se crea la tabla.

MySQL almacena los valores SET numéricamente, con el bit menos significativo del valor almacenado correspondiente al primer miembro del SET. Si recuperas un valor SET en un contexto numérico, el valor recuperado tiene los bits del set correspondiente a los miembros del SET que tomó el valor de la columna. Por ejemplo, puedes recuperar los valores numéricos de una columna SET como esta:

```
mysql> SELECT set_col+0 FROM tbl_name;
```

Si se almacena un número en una columna SET, los bits situados en la representación binaria del número determinan los miembros de la lista en el valor de la columna. Supón que una columna se especifica como SET(“a”, “b”, “c”, “d”). Entonces los miembros tienen los siguientes valores binarios:

Miembro SET	Valor decimal	Valor binario
a	1	0001
b	2	0010
c	4	0100
d	8	1000

Si asignas un valor de 9 a esta columna, el valor binario es 1001, con lo que resulta que los valores seleccionados del set son “a,d”.

Para un valor que contiene más de un elemento del SET, no importa el orden en el que se inserten. Tampoco importa cuantas veces un elemento dado se liste en el valor. Cuando un valor es recuperado posteriormente, cada elemento en el valor aparecerá una vez, con el orden de los elementos de acuerdo con el listado especificado en el momento de la creación de la tabla. Por ejemplo, si la columna es especificada como SET(“a”,”b”,”c”,”d”), entonces “a,d” y “d,a,a,d” apareceran como “a,d” al ser recuperados.

Si indicas a la columna SET un valor no soportado, el valor será ignorado.

Los valores SET se ordenan numéricamente. Los valores NULL se ordenan antes que los valores NOT NULL.

Normalmente puedes realizar una sentencia SELECT en una columna SET utilizando el operador LIKE o la función FIND_IN_SET():

```
mysql> SELECT * FROM tbl_name WHERE set_col LIKE “%value%”;
mysql> SELECT * FROM tbl_name WHERE FIND_IN_SET(‘value’,set_col)>0;
```

Pero lo siguiente también funcionará:

```
mysql> SELECT * FROM tbl_name WHERE set_col=’val1,val2’;
mysql> SELECT * FROM tbl_name WHERE set_col & 1;
```

La primera de estas sentencias busca una coincidencia exacta. El segundo mira los valores que contienen el primer elemento del set [realiza una comparación bit a bit].

Si quieres tener todos los valores posibles de una columna SET, puedes usar SHOW COLUMNS FROM table_name LIKE set_column_name y interpretar la definición del SET de la segunda columna.

6.2.4. Eligiendo el tipo de columna correcto

Para un uso más eficiente de almacenamiento, trata de utilizar los tipos más precisos en cada caso. Por ejemplo, si una columna integer se utilizará para valores en el rango entre 1 y 99999, el tipo MEDIUMINT UNSIGNED es el más adecuado.

La representación ajustada de los valores monetarios es un problema habitual. En MySQL, deberías utilizar el tipo DECIMAL. Se almacena como cadena, de modo que no se pierde ajuste. Si el ajuste no es muy importante, el tipo DOUBLE puede ser suficientemente bueno.

Para una gran precisión, siempre puedes convertir un tipo de coma fija almacenado en un BIGINT. Ello te permite hacer todos los cálculos con enteros y convertir luego los valores a coma flotante sólo en caso de necesidad.

6.2.5. Utilizando tipos de columnas para otros motores de bases de datos

Para simplificar el uso del código escrito en implementaciones de SQL de otros vendedores, MySQL mapea los tipos de columnas como se presenta en la tabla siguiente. Estos mapeos simplifican mover definiciones de tablas desde otros motores de bases de datos a MySQL:

Tipos de otros sistemas	Tipo MySQL
BINARY(NUM)	CHAR(NUM) BINARY
CHAR VARYING(NUM)	VARCHAR(NUM)
FLOAT4	FLOAT
FLOAT8	DOUBLE
INT1	TINYINT
INT2	SMALLINT
INT3	MEDIUMINT
INT4	INT
INT8	BIGINT
LONG VARBINARY	MEDIUMBLOB
LONG VARCHAR	MEDIUMTEXT
MIDDELINT	MEDIUMINT
VARBINARY(NUM)	VARCHAR(NUM) BINARY

El mapeo de los tipos de columna tiene lugar en el momento de la creación de la tabla. Si creas una tabla con tipos usados por otros sistemas y entonces indicas una sentencia DESCRIBE tbl_name, MySQL informa sobre la estructura de la tabla utilizando los tipos MySQL equivalentes.

6.2.6. Requerimientos de almacenamiento de cada tipo de columna

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

Tipos numéricos

Tipo de columna	Almacenamiento requerido (bytes)
TINYINT	1
SMALLINT	2
MEDIUMINT	3
INT	4
INTEGER	4
BIGINT	8
FLOAT(X)	4 si X<=24 8 si 25<=X<=53
FLOAT	4
DOUBLE	8
DOUBLE PRECISION	8
REAL	8
DECIMAL(M,D)	M+2 si D>0 M+1 si D=0 D+2 si M<D
NUMERIC(M,D)	M+2 si D>0 M+1 si D=0 D+2 si M<D

Tipos de fecha y hora:

Tipo de columna	Alamcenamiento requerido (bytes)
DATE	3
DATETIME	8
TIMESTAMP	4
TIME	3
YEAR	1

Tipos de cadena

Tipo de columna	Alamcenamiento requerido (bytes)
CHAR(M)	M, si 1<=M<=255
VARCHAR(M)	L+1, donde L<=M y 1<=M<=255
TINYBLOB,TINYTEXT	L+1, donde $L<2^8$
BLOB,TEXT	L+2, donde $L<2^{16}$
MEDIUMBLOB,MEDIUMTEXT	L+3, donde $L<2^{24}$
LONGBLOB,LONGTEXT	L+4, donde $L<2^{32}$
ENUM('valor1', 'valor2',...)	1 o 2, dependiendo del número de enumeraciones (máx 65536 valores)
SET('valor1', 'valor2',...)	1, 2, 3, 4, u 8, dependiendo del número de miembros (máx 64 miembros)

VARCHAR y los tipos BLOB y TEXT son tipos de longitud variable, para los cuales los requerimientos de almacenamiento dependen de la longitud actual de los valores de las columnas (representados por L en la tabla precedente), más que en en el tamaño máximo posible de los valores. Por ejemplo, un campo VARCHAR(10) puede almacenar una cadena con una longitud máxima de 10 caracteres. El almacenamiento actual requerido es la longitud de la cadena (L), más 1 byte para almacenar la longitud de la cadena. Para la cadena 'abcd', L es 4 y el requerimiento de almacenamiento es de 5 bytes.

Los tipos BLOB y TEXT requieren entre 1 y 4 bytes para almacenar la longitud del valor de la columna, dependiendo de la longitud máxima posible del tipo. Puedes ver la sección 6.2.3.2. [BLOB].

Si una tabla incluye un tipo de columna de longitud variable, el formato será de longitud variable. Cabe anotar que cuando una tabla se crea, MySQL puede, bajo ciertas condiciones, cambiar una columna de un tipo de longitud variable a uno de longitud fija, o viceversa. Puedes ver la sección 6.5.3.1. [Silent column changes].

El tamaño de un objeto ENUM se determina por el número de valores diferentes enumerados. Un byte se utiliza por enumeraciones de hasta 255 elementos. Se utilizan dos bytes en los casos de hasta 65536 elementos enumerados. Puedes ver la sección 6.2.3.3. [ENUM]

El tamaño de un objeto SET se determina por el número de miembros diferentes del set. Si el tamaño del set es N, el objeto ocupa (N+7)/8 bytes, redondeado hacia arriba en 1, 2, 3, 4 o 8 bytes. Un SET Puede tener un máximo de 64 miembros. Puedes ver la sección 6.2.3.4 [SET].

6.3. Funciones a utilizar en las cláusulas SELECT y WHERE

Una expresion_select o definicion_where en una sentencia SQL puede consistir en cualquier expresión que utilice las funciones descritas más adelante.

Una expresión que contiene NULL siempre produce un valor NULL a menos que se indique otro caso en la documentación para los operadores y funciones implicados en la expresión.

Nota: No debe haber espacios en blanco entre el nombre de la función y el paréntesis de los parámetros. Esto ayuda al intérprete de MySQL a distinguir entre llamadas a funciones y referencias a tablas o columnas que tengan el mismo nombre que la función. Los espacios alrededor de los argumentos son permitidos, a pesar de ello.

Puedes forzar a MySQL a aceptar espacios después del nombre de la función iniciando mysqld con –-ansi o usando CLIENT_IGNORE_SPACE en mysql_connect(), pero en este caso, todos los nombres de funciones se convierten en palabras reservadas. Puedes ver la sección 1.7.2. [ANSI mode].

Por exigencias de brevedad, los ejemplos visualizan el resultado del programa mysql en forma abreviada. Por lo tanto esto:

```
mysql> SELECT MO(29,9);
1 rows in set (0.00 sec)
```

mod(29,9)
2

se visualiza así:

```
mysql> SELECT MOD(29,9);
        ->2
```

6.3.1. Operadores y funciones independientes del tipo de datos

6.3.1.1. Paréntesis

(...)

Utiliza paréntesis para forzar el orden de evaluación en una expresión. Por ejemplo:

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

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

6.3.1.2. Operadores de comparación

Las operaciones de comparación dan como resultados los valores 1 (TRUE), 0 (FALSE) o NULL. Estas funciones actúan tanto para números como para cadenas. Las cadenas se convierten automáticamente a números y los números a cadenas según necesidad (como en Perl).

MySQL realiza comparaciones utilizando las siguientes reglas:

- Si uno o ambos de los argumentos son NULL, el resultado de la comparación es NULL, excepto para el operador <=>.
- Si ambos argumentos en una operación de comparación son cadenas, se comparan como cadenas.
- Si dos argumentos son enteros, se comparan como enteros.
- Valores haxadecimales son tratados como cadenas binarias si no se comparan como números.
- Si uno de los argumentos es una columna TIMESTAMP o DATETIME y el otro argumento es una constante, la constante se convierte a timestamp antes de realizar la comparación. Esto se hace para ser mas amigable a ODBC.
- En el resto de casos, los argumentos se comparan como números de coma flotante.

Por defecto, las comparaciones de cadenas se hacen ignorando el uso de mayúsculas/minúsculas utilizando el juego de caracteres actuales (ISO-8859-1 Latin1 por defecto, que funciona excelentemente en inglés).

Los siguientes ejemplos ilustran la conversión de cadenas a números para las operaciones de comparación:

```
mysql> SELECT 1> '6x';
        -> 0

mysql> SELECT 7> '6x';
        -> 1

mysql> SELECT 0> 'x6';
        -> 0

mysql> SELECT 0 = 'x6';
        -> 1
```

= Igualdad:

```
mysql> SELECT 1 = 0;
        -> 0
```

```
mysql> SELECT '0'  = 0;
        -> 1

mysql> SELECT '0.0'  = 0;
        -> 1

mysql> SELECT '0.01'  = 0;
        -> 0

mysql> SELECT '.01'  = '0.01';
        -> 1
```

<>
!= (diferente, no igual)

```
mysql> SELECT '.01'  <> '0.01';
        -> 1
mysql> SELECT .01 <> '0.01';
        -> 0
mysql> SELECT 'zapp'  <> 'zappp';
        -> 1
```

<= Menor o igual que:

```
mysql> SELECT 0.1 <> 2;
        -> 1
```

< Menor que:

```
mysql> SELECT 2 <> 2;
        -> 0
```

>= Mayor o igual que:

```
mysql> SELECT 2>=2;
        -> 0
```

> mayor que:

```
mysql> SELECT 2>2;
        -> 0
```

<=> Igual a incluyendo valores NULL

```
mysql> SELECT 1 <=> 1, NULL <=> NULL, 1<=>NULL;
        -> 1 1 0
```

IS NULL
IS NOT NULL

Comprueba si el valor es no no NULL:

```
mysql> SELECT 1 IS NULL, 0 IS NULL, NULL IS NULL;
```

```
        -> 0 0 1

mysql> SELECT 1 IS NOT NULL, 0 IS NOT NULL, NULL IS NOT NULL;
        -> 1 1 0
```

Para poder trabajar bien con otros programas, MySQL soporta las siguientes características al utilizar IS NULL:

- Puedes encontrar la última fila insertada con:

```
SELECT * FROM tbl_name WHERE auto_col IS NULL
```

Esto puede desactivarse indicando SQL_AUTO_IS_NULL=0. Puedes ver la sección 5.5.6 [SET OPTION].

- Para columnas DATE y DATETIME NOT NULL, puedes encontrar el valor especial de fecha 0000-00-00 utilizando:

  ```
  SELECT * FROM tbl_name WHERE date_column IS NULL
  ```

Esto es necesario para que algunas funciones ODBC funcionen (ya que ODBC no soporta la fecha 0000-00-00).

expr BETWEEN min AND max

Si expr es mayor que o igual a min y expr es menor o igual a max, BETWEEN retorna 1, y en cualquier otro caso retorna 0. Esto es equivalente a la expresión (min <= expr AND exp <=max) si todos los argumentos son del mismo tipo. El primer argumento (expr) determina como se realiza la comparación, como sigue:

- Si expr es una columna TIMESTAMP, DATE o DATETIME, MIN() y MAX() se formatean al mismo formato si son constantes.
- Si expr es una expresión de cadena que no diferencia mayúsculas y minúsculas, se hace una comparación case-insensitive.
- Si expr es una expresión de cadena que diferencia mayúsculas y minúsculas, se hace una comparación case-sensitive.
- Si expr es una expresión entera, se realiza una comparación de valores enteros.
- En el resto de casos, se realiza una comparación de coma flotante (real).

```
mysql> SELECT 1 BETWEEN 2 AND 3;
        -> 0
mysql> SELECT 'b' BETWEEN 'a' AND 'c';
        -> 1
mysql> SELECT 2 BETWEEN 2 AND '3';
        -> 1
mysql> SELECT 2 BETWEEN 2 AND 'x-3';
        -> 0
```

expr NOT BETWEEN min AND max

Equivale a NOT(expr BETWEEN min AND max).

expr IN(valor, ...)

Retorna 1 si expr es cualquiera de los valores en la lista IN, y en caso contrario retorna 0. Si todos los valores son constantes, los valores se evalúan de acuerdo con el tipo de expr y se ordenan. La búsqueda del elemento se realiza con una búsqueda dicotómica (binary search). Esto implica que IN es muyt rápido si la lista de valores consiste enteramente de constantes. Si expr es una expresión de cadena que diferencia mayúsculas y minúsculas, la comparación de la cadena se realiza en un modo case-sensitive.

```
mysql> SELECT 2 IN (0,3,5,'wefwf');
        -> 0

mysql> SELECT 'wefwf' IN (0,3,5,'wefwf');
        -> 1
```

expr NOT IN (valor,...)

Equivalente a NOT(expr IN (valor,...))

ISNULL(expr)

Si expr es NULL, ISNULL() retorna 1, y en el resto de casos retorna 0.

```
mysql> SELECT ISNULL(1+1);
        -> 0

mysql> SELECT ISNULL(1/0);
        -> 1
```

Cabe anotar que una comparación de valores NULL siempre será falsa.

COALESCE(list)

Retorna el primer elemento que no equivale a NULL en la lista:

```
mysql> SELECT COALESCE(NULL,1);
        -> 1
mysql> SELECT COALESCE(NULL, NULL, NULL);
        -> NULL
```

INTERVAL(N,N1,N2,N3,...)

Retorna si N<N1, 1 si N<N2, y continúa así. Todos los argumentos son tratados como enteros. Se requiere que N1<N2<N3<...<Nn para que esta función actúe correctamente. Ello es debido a que se utiliza una busqueda dicotómica (binary search), muy rápida:

```
mysql> SELECT INTERVAL(23,1,15,17,30,44,200);
        -> 3

mysql> SELECT INTERVAL(10,1,10,100,1000);
        -> 2

mysql> SELECT INTERVAL(22,23,30,44,200);
        -> 0
```

Si estás comparando cadenas case-sensitive con cualquiera de los operadores estándares (=,<>,..., pero no LIKE) los espacios finales (espacios, tabulaciones y saltos de línea) se ignorarán.

```
mysql> SELECT “a ”= “A \n”;
        -> 1
```

6.3.1.3. Operadores lógicos

Todos los operadores lógicos evalúan a 1 (TRUE), 0 (FALSE) o NULL (Desconocido, lo que en la mayoría de casos equivale a FALSE).

Operador	Explicación
NOT, !	No lógico. Evalúa como 1 si el operando es 0, y en cualquier otro caso evalúa a 0. Excepción: NOT NULL evalúa a NULL: mysql> SELECT NOT 1; ->0 mysql> SELECT NOT NULL; ->NULL mysql> SELECT ! (1+1); ->0 mysql>SELECT ! 1+1; -> 1
OR, ||	O lógico. Evalúa como 1 tanto si el operando no es 0 ni NULL: mysql> SELECT 1 || 0; ->1 mysql> SELECT 0 ||0; ->0 mysql> SELECT 1 || NULL; ->1
AND, &&	Y lógico. Para operandos no nulos, evalúa como 1. Si ambos operandos son diferentes de 0, y a 0 en cualquier otro caso. Produce NULL si cualquier operando es NULL. mysql> SELECT 1 && 1; ->1 mysql> SELECT 1 && 0; ->0 mysql> SELECT 1 && NULL; ->NULL
XOR	Operador O-exclusivo. Para operandos no nulos, evalúa como 1 si sólo uno de los operadores es diferente de cero. Produce NULL si cualquier operando es NULL: mysql> SELECT 1 XOR 1; ->0 mysql> SELECT 1 XOR 0; ->1 mysql> SELECT 1 XOR NULL; ->NULL

6.3.1.4. Funciones de control de flujo

IFNULL(expr1, expr2)

Si expr1 no es NULL, IFNULL() retorna expr2, y en caso contrario retorna expr1. IFNULL() retorna un valor de cadena o número, dependiendo del contexto en el que se utiliza:

```
mysql> SELECT IFNULL(1,0);
->1

mysql> SELECT IFNULL(NULL,10);
->10

mysql> SELECT IFNULL(1/0,10);
->10

mysql> SELECT IFNULL(1/0,'yes');
->'yes'
```

NULLIF(expr1,expr2)

Si expr1=expr2 es cierto, retorna NULL, y sinó, retorna expr1. Esto es lo mismo que CASE WHEN x=y THEN NULL ELSE x END:

```
mysql> SELECT NULLIF(1,1);
->NULL

mysql> SELECT NULLIF(1,2);
->1
```

expr1 se evalúa dos veces en MySQL si los argumentos son iguales.

IF(expr1,expr2,expr3)

Si expr1 es TRUE (expr1<>0 y expr1<>NULL) entonces IF() retorna expr2, y sinó retorna expr3. IF() retorna un valor numérico o de cadena, depediendo del contexto en el que se utilice.

```
mysql> SELECT IF(1>2,2,3);
->3

mysql> SELECT IF(1<2,'yes','no');
->'yes'

mysql> SELECT IF(SCRCMP('text','text1'),'no', 'yes');
->'no'
```

expr1 se evalúa como un valor entero, esto significa que si estás testeando un valor de coma flotante o una cadena, debes hacerlo utilizando una operación de comparación.

```
mysql> SELECT IF(0.1,1,0);
->0
```

```
mysql> SELECT IF(0.1<>0,1,0);
->1
```

En el primer caso, IF(0.1) retorna 0 porque 0.1 es convertido a un valor entero, resultando que se testea IF(0). Puede ser que no es lo que esperaras. En el segundo caso, se compara el valor original de coma flotante para ver si es cero. El resultado de esta comparación se usa como entero.

El valor de retorno por defecto de IF() (que puede importar cuando se almacena en una tabla temporal) se calcula en la versión 3.23 de MySQL como sigue:

Expresión	valor de retorno
expr2 o expr3 retorna cadena	cadena
expr2 o expr3 retorna valor de coma flotante	coma flotante
expr2 o expr3 retorna entero	entero

Si expr2 y expr3 son cadenas, entonces el resultado es case-sensitive si las dos cadenas son case-sensitive (a partir de la versión 3.23.51).

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 cuando value=compare-value. La segunda versión retorna el resultado de la primera condición, que es cierta. Si no hubiera ningún resultado coincidente, se retorna el resultado que hay después de ELSE. Si no hay ninguna parte ELSE, se retorna NULL:

```
mysql>SELECT CASE 1 WHEN 1 THEN “one” WHEN 2 THEN “two” ELSE “more” END;
->“one”

mysql>SELECT CASE WHEN 1>0 THEN “true” ELSE “false” END;
->“true”

mysql>SELECT CASE BINARY “B” WHEN “a” THEN 1 WHEN “b” THEN 2 END;
->NULL
```

El tipo de valor de retorno (INTEGER, DOUBLE o STRING) es el mismo que el tipo del primer valor retornado (la expresión después del primer THEN).

6.3.2. Funciones de cadena

Las funciones de valor de cadena retornan NULL si la longitud del resultado superara el valor del parámetro de servidor max_allowed_packet. Puedes ver la sección 5.5.2 [Serveer parameters].

ASCII(str)

Retorna el valor del código ASCII del carácter más a la izquierda de la cadena str. Retorna 0 si str es una cadena vacía. Retorna NULL si str es NULL:

```
mysql> SELECT ASCII(‘2’);
->50
```

```
mysql> SELECT ASCII(2);
->50

mysql> SELECT ASCII('dx');
->100
```

Puedes ver también la función ORD().

ORD(str)

Si el carácter más a la izquierda de la cadena str es un carácter multi-byte, retorna el código para este carácter, calculado como el valor en código ASCII para sus caracteres constituyentes utilizando la fórmula: ((primer código ASCII)*256+(segundo código ASCII)) [*256+(primer código ASCII)*256]. Si el carácter más a la izquierda no es un carácter multi-byte, retorna el mismo valor que la función ASCII hace:

```
mysql> SELECT ORD('2');
->50
```

CONV(N,from_base,to_base)

Convierte números entre diferentes bases. Retorna una representación de cadena del número N, convertido de la base from_base a la base to_base. Retorna NULL si el argumento es NULL. El argumento N es interpretado como un entero, pero puede ser especificado como un entero o una cadena. La base mínima es 2 y la máxima es 36. Si to_base es un valor negativo, N es tratado como un valor con signo. En caso contrario, se trata como valor unsigned. CONV trabaja con precisión de 64 bits.

```
mysql> SELECT CONV("a",16,2);
->'1010'

mysql> SELECT CONV("6E",18,8);
->'172'

mysql> SELECT CONV("-17",10,-18);
->'-H'
mysql> SELECT CONV(10+"10"+0xa,10,10);
->'40'
```

BIN(N)

Retorna una representación de cadena con el valor binario de N, donde N es un número largo (BIGINT). Esto es equivalente a CONV(N,10,8). Retorna NULL si N es NULL:

```
mysql> SELECT BIN(12);
->'1100'
```

OCT(N)

Retorna una representación de cadena con el valor octal de N, donde N es un número largo (BIGINT). Esto es equivalente a CONV(N,10,8). Retorna NULL si N es NULL:

```
mysql> SELECT OCT(12);
->‘14’
```

HEX(N_or_S)

Si N_or_S es número, retorna una representación de cadena del valor hexadecimal de N, donde N es un entero largo (BIGINT). Esto es equivalente a CONV(N,10,16).

Si N_or_S es una cadena, retorna cadena hexadecimal de N_or_S donde cada carácter en N_or_S se convierte a 2 dígitos hexadecimales. Esto es el inverso de las cadenas 0xff.

```
mysql> SELECT HEX(255);
->‘FF’
mysql> SELECT HEX(“abc”);
->‘61623’
mysql> SELECT 0x61623;
->“abc”
```

CHAR(N,...)

CHAR() interpreta los argumentos como enteros y retorna una cadena consistente en los caracteres dados por el código ASCII de estos enteros. Los valores NULL son saltados:

```
mysql> SELECT CHAR(77,121,83,81,’76’);
->‘MySQL’
mysql> SELECT CHAR(77,77.3,’77.3’);
->‘MMM’
```

CONCAT(str1,str2,...)

Retorna la cadena que resulta de concatenar los argumentos. Retorna NULL si cualquier argumento es NULL. Puede tener más de 2 argumentos. Un argumento numérico se convierte al formato de cadena equivalente:

```
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 y es un formato especial de CONCAT(). El primer argumento es el separador del resto de los argumentos. El separador puede ser una cadena como el resto de los argumentos. Si el separador es NULL, el resultado será NULL. La función saltará cualquier valor NULL y cadenas vacías, después del argumento separador. El separador se añadirá entre las cadenas a concatenar:

```
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’
```

LENGTH(str)
OCTET_LENGTH(str)
CHAR_LENGTH(str)
CHARACTER_LENGTH(str)

Retornan la longitud de la cadena str:

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

Cabe anotar que para CHAR_LENGTH() y CHARACTER_LENGTH(), los caracteres multi-byte se cuentan una sola vez.

BIT_LENGTH(str)

Retorna la cadena de la cadena str en bits:

```
mysql> SELECT BIT_LENGTH('text');
->32
```

LOCATE(substr,str)
POSITION(substr IN str)

Retorna la posición de la primera ocurrencia de la subcadena substr en la cadena str. Retorna 0 si substr no se halla en str.

```
mysql> SELECT LOCATE('bar', 'foobarbar');
->4
mysql> SELECT LOCATE('xbar', 'foobarbar');
->4
```

Esta función soporta multi-byte. En MySQL 3.23 esta función es case sensitive, mientras que en 4.0 sólo es case sensitive si cualquier argumento es una cadena binaria.

LOCATE(substr,str,pos)

Retorna la posición de la primera ocurrencia de la subcadena substr en la cadena str, empezando en la posición pos. Retorna 0 si substr no se halla en str.

```
mysql> SELECT LOCATE('bar', 'foobarbar',5);
->7
```

Esta función soporta el multi-byte. En MySQL 3.23 esta función es case-sensitive, mientras que en 4.0 sólo lo es si se da el caso caso que algún argumento sea una cadena binaria.

INSTR(str,substr)

Retorna la posición de la primera ocurrencia de la subcadena substr en la cadena str. Esto es lo mismo que la forma de dos argumentos LOCATE(), excepto en que los argumentos son intercambiados:

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

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

Esta función soporta multi-byte. En MySQL 3.23 esta función es case sensitive, mientras que en 4.0 sólo lo es en caso que algún argumento sea una cadena binaria.

LPAD(str,len,padstr)

Retorna la cadena str, rellenada por la izquierda con la cadena padstr hasta que str tenga len caracteres de largo. Si str es más largo que len entonces se recortará para que tenga len caracteres.

```
mysql> SELECT LPAD('hi',4, '??');
->'??hi'
```

RPAD(str,len,padstr)

Retorna la cadena str, rellenada por la derecha con la cadena padstr hasta que str tiene len caracteres de largo. Si str es más larga que len entonces será recortada a len caracteres.

```
mysql> SELECT LPAD('hi',5, '??');
->'hi???'
```

LEFT(str,len)

Retorna los len caracteres más a la izquierda de la cadena str.

```
mysql> SELECT LEFT('foobarbar',5);
->'fooba'
```

Esta función soporta multi-byte.

RIGHT(str,len)

Retorna los len caracteres más a la derecha de la cadena str:

```
mysql> SELECT RIGHT('foobarbar',4);
->'rbar'
```

Esta función soporta multi-byte.

SUBSTRING(str,pos,len)
SUBSTRING(str FROM pos FOR len)
MID(str,pos,len)

Retorna la subcadena de len caracteres de largo desde la cadena str, empezando desde la posición pos. La forma variante que usa FROM es la sintaxis ANSI SQL 92:

```
mysql> SELECT SUBSTRING('Quadratically',5,6);
```

->'ratica'

Esta función soporta multi-byte.

SUBSTRING(str,pos)
SUBSTRING(str FROM pos)

Retorna una subcadena desde la cadena str empezando desde la posición pos:

mysql> SELECT SUBSTRING('Quadratically',5);
->'ratically'

mysql> SELECT SUBSTRING('foobarbar' FROM 4);
->'barbar'

Esta función soporta multi-byte

SUBSTRING_INDEX(str,delim,count)

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

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

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

Esta función soporta multi-byte.

LTRIM(str)

Retorna la cadena str con los espacios precedentes eliminados:

mysql> SELECT LTRIM(' barbar');
->'barbar'

Esta función soporta multi-byte.

RTRIM(str)

Retorna la cadena str con los espacios finales eliminados:

mysql> SELECT RTRIM('barbar ');
->'barbar'

Esta función soporta multi-byte.

TRIM([[BOTH|LEADING|TRAILING] [remstr] FROM] str)

Retorna la cadena str con todos los prefijos remstr y/o sufijos eliminados. Si no se indica ninguno de los especificadores BOTH, LEADING o TRAILING, se asume BOTH. Si no se especifica remstr, 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 'x' FROM 'xxxbarxxx');
->'xxxbar'
```

Esta función soporta multibyte.

SOUNDEX(str)

Retorna una cadena soundex de str. Dos cadenas que pueden sonar casi igual deberían tener cadenas soundex idénticas. Una cadena estándar soundex tiene 4 caracteres de largo, pero la función SOUNDEX() retorna una cadena arbitrariamente larga. Puedes usar SUBSTRING() en el resultado para conseguir una cadena soundex estándar. Todos los caracteres que no sean alfanuméricos se ignoran en la cadena dada. Todos los caracteres internacionales alfanuméricos fuera del rango A-Z son tratados como vocales.

```
mysql> SELECT SOUNDEX('Hello');
->'H400'
mysql> SELECT SOUNDEX('Quadratically');
->'Q36324'
```

SPACE(N)

Retorna una cadena consistente en N caracteres de espacio:

```
mysql> SELECT SPACE(6);
->'      '
```

REPLACE(str,from_str,to_str)

Retorna la cadena str con todas las ocurrencias de la cadena from_str reemplazada por la cadena to_str:

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

Esta función soporta multi-byte.

REPEAT(str,count)

Retorna una cadena consistente en la cadena str repetida count veces. Si count <=0, retorna una cadena vacía. Retorna NULL si str o count son nulos:

```
mysql> SELECT REPEAT('MySQL',3);
->'MySQLMySQLMySQL'
```

REVERSE(str)

Retorna la cadena str con el order de caracteres invertido:

```
mysql> SELECT REVERSE('abc');
->'cba'
```

La función soporta multi-byte.

INSERT(str,pos,len,newstr)

Retorna la cadena str, con la subcadena iniciando en la posición pos y len caracteres de largo remplazados por la cadena newstr:

```
mysql> SELECT INSERT('Quadratic',3,4,'What');
->'QuWhattic'
```

Esta función soporta multi-byte.

ELT(N,str1,str2,str3,...)

Retorna str si N=1, str2 si N=2, y así hasta el final. Retorna NULL si N es menor que 1 o mayor que el número de argumentos. ELT() es el complemento de FIELD():

```
mysql> SELECT ELT(1,'ej','Heja','hej','foo');
->'ej'

mysql> SELECT ELT(4,'ej','Heja','hej','foo');
->'foo'
```

FIELD(str,str1,str2,str3,...)

Retorna el índice de str en la lista str1,str2,str3,... Retorna 0 si str no se encuentra. FIELD() es el complement de ELT():

```
mysql> SELECT FIELD('ej','Hej','ej','hej','foo');
->2
mysql> SELECT FIELD('fo','Hej','ej','hej','foo');
->0
```

FIND_IN_SET(str,strlist)
Retorna un valor entre 1 y N si la cadena str está en la lista strlist consistente en N subcadenas. Una lista de cadenas es una cadena compuesta de subcadenas separadas por caracteres ',' . Si el primer argumento es una cadena constante y el segundo es una columna del tipo SET, la función FIND_IN_SET() se optimiza al usar aritmética de bits! Retorna 0 si str no se halla en strlist o si strlist es una cadena vacía. Retorna NULL si algún argumento es NULL. Esta función no funcionará propiamente si el primer argumento contiene una ',':

```
mysql> SELECT FIND_IN_SET('b','a,b,c,d');
->2
```

MAKE_SET(bits,str1,str2,...)

Retorna un conjunto [set] (una cadena que contiene subcadenas separadas por caracteres ‘,’) consistente en las cadenas que se corresponden con el conjunto de bits. str1 corresponde al bit 0, str2 al bit 1, etc. Las cadenas NULL de str1, str2,... no se añaden al resultado:

```
mysql> SELECT MAKE_SET(1,'a','b','c');
->'a'
mysql> SELECT MAKE_SET(1|4,'hello','nice','world');
->'hello,world'
mysql> SELECT MAKE_SET(0,'a','b','c');
->''
```

EXPORT_SET(bits, on,offf, [separator,[number_of_bits]])

Retorna una cadena donde cada bit indicado en ‘bit’ da una cadena ‘on’ y por cada bit a cero consigues una cadena ‘off’. Cada cadena se separa con ‘separator’ (default ‘,’) y sólo ‘number_of_bits’ (64 por defecto) de bits usados:

```
mysql> SELECT EXPORT_SET(5,'Y','N',  ',' ,4);
->'Y,N,Y,N'
```

LCASE(str)
LOWER(str)

Retorna la cadena str con todos los caracteres cambiados a minúsculas de acuerdo con el actual mapeo del juego de caracteres (por defecto ISO-8859-1 Latin1):

```
mysql> SELECT LCASE('QUADRATICALLY');
->'quadratically'
```

Esta función soporta multi-byte.

UCASE(str)
UPPER(str)

Retorna la cadena str con todos los caracteres cambiados a mayúscula de acuerdo con el actual mapeo de juego de caracteres (por defecto ISO-8859-1 Latin1):

```
mysql> SELECT UCASE('Hej');
->'HEJ'
```

Esta funciópn soporta multi-byte.

LOAD_FILE(file_name)

Lee un archivo y retorna su contenido como cadena. El archivo debe estar en el servidor, debes especificar la ruta completa hasta él, y debes disponer del privilegio FILE. El archivo debe ser leíble por todos y ser más pequeño que max_allowed_packet.

Si el archivo no existe o no puede ser leído debido a alguna de las razones anteriores, la función retorna NULL:

```
mysql> UPDATE tbl_name
```

```
SET blob_column=LOAD_FILE("/tmp/picture")
WHERE id=1;
```

Si no estás utilizando la versión 3.23 de MySQL, debes realizar la lectura del archivo desde el interior de tu aplicación y crear una sentencia INSERT para actualizar la base de datos con la información del archivo. Una forma de hacer esto, estás usando la librería MySQL++, que puede ser hallada http://www.mysql.com/documentation/mysql++/mysql++-examples.html

MySQL convierte automáticamente números a cadenas según necesidad, y viceversa:

```
mysql> SELECT 1+"1";
->2

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

Si quieres convertir un número a cadena explícitamente, pásalo como argumento a CONCAT(). Si una cadena es dada binaria como argumento, la cadena resultante es también una cadena binaria. Un número convertido a una cadena es tratada como cadena binaria. Esto sólo afecta a las comparaciones.

6.3.2.1. Funciones de comparación de cadenas

Normalmente, si cualquier expresión en una comparación de cadenas es case-sensitive, la comparación se realiza en modo case-sensitive.

expr LIKE pat [ESCAPE 'escape-char']

Búsqueda de patrones utilizando expresiones regulares simples de SQL. Retorna 1 (TRUE) o 0 (FALSE). Con LIKE puedes utilizar los dos siguientes caracteres comodín en el patrón:

Carácter	Descripción
%	Se ajusta a cualquier número de caracteres, incluso 0 caracteres.
_	Se ajusta a exactamente un carácter.

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

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

Para comprobar ejemplos de un carácter comodín, precédelo con el carácter de escape. Si no especificas el el carácter ESCAPE, se asume '\':

Carácter	Descripción
\%	Se ajusta a cualquier número de caracteres, incluso 0 caracteres.
_	Se ajusta a exactamente un carácter.

```
mysql> SELECT 'David!' LIKE 'David\_';
->0

mysql> SELECT 'David_' LIKE 'David\_';
->1
```

Para especificar un carácter diferente, utiliza la cláusula ESCAPE:

```
mysql> SELECT 'David_' LIKE 'David|_' ESCAPE '|';
->1
```

Las dos sentencias siguientes ilustran que las comparaciones de cadenas son case-insensitive a menos que uno de los operandos sea cadena binaria:

```
mysql> SELECT 'abc' LIKE 'ABC';
->1

mysql> SELECT 'abc' LIKE BINARY 'ABC';
->0
```

Se permite LIKE en expresiones numéricas! (Esto es una extensión de MySQL respecto el LIKE del ANSI SQL):

```
mysql> SELECT 10 LIKE '1%';
->1
```

Nota: Debido a que MySQL utiliza la sintaxis de escape de C en cadenas (por ejemplo, '\n'), debes duplicar cualquier '\' que utilices en tus cadenas LIKE. Por ejemplo, para encontrar '\n', especifícalo como '\\n'. Para encontrar '\', especifícalo como '\\\\' (se necesita una contrabarra de escape por cada contrabarra literal).

expr NOT LIKE [EXCAPE 'escape-char']

Equivalente a NOT (expr LIKE pat [ESCAPE 'escape-char']).

expr REGEXP pat
expr RLIKE pat

Realiza una búsqueda de un patrón de expresión cadena expr contra un patrón pat. El patrón puede ser una expresión regular extendida. Puedes ver Apéndice G [Regexp]. Retorna 1 si expr se ajusta a pat. En caso contrario, retorna 0. RLIKE es un sinónimo para REGEXP, aportado para compatibilizar con mSQL.

Nota: Debido a que MySQL utiliza la sintaxis de escape de C en cadenas (por ejemplo, '\n'), debes duplicar cualquier '\' que utilices en tus cadenas REGEXP. REGEXP, desde la versión 3.23.4 de MySQL, es case-insensitive para cadenas normales (no binarias):

```
mysql> SELECT 'Monty!' REGEXP 'm%y%%';
->0

mysql> SELECT 'Monty!' REGEXP '.*';
->1

mysql> SELECT 'new*\n*line' REGEXP 'new\\*.\\*line';
->1

mysql> SELECT "a" REGEXP "A", "a" REGEXP BINARY "A" ;
->1 0
```

```
mysql> SELECT "a" REGEXP "^[a-d]" ;
->1
```

REGEXP y RLIKE utilizan el actual juego de caracteres (ISO-8859-1 Latin1 por defecto), al decidir el tipo de carácter.

expr NOT REGEXP pat
expr NOT RLIKE pat

Equivalente a NOT(expr REGEXP pat)

STRCMP(expr1,expr2)

STRCMP() retorna 0 si las cadenas son iguales, -1 si el primer argumento es menor que el segundo de acuerdo con el número de orden, y 1 en el resto de casos:

```
mysql> SELECT STRCMP('text1', 'text2');
->-1

mysql> SELECT STRCMP('text2', 'text');
->1

mysql> SELECT STRCMP('text', 'text');
->0
```

MATCH(col1,col2,...) AGAINST (expr)
MATCH(col1,col2,...) AGAINST (expr IN BOOLEAN MODE)

MATCH...AGAINST() se utiliza para búsqueda de texto completo y retorna la relevancia – medida de similitud entre el texto en las columnas (col1,col2,...) y la consulta expr. La relevancia es un valor de coma flotante positivo. La relevancia 0 significa que no hay similitud. MATCH...AGAINST() está disponible en MySQL 3.23.23 o superior. La extensión IN BOOLEAN MODE fue añadida en la versión 4.0.1. Para detalles y ejemplos uso puedes ver la sección 6.8 [Fulltext search].

6.3.2.2. Case-Sensitivity [Diferenciación mayúsculas-minúsculas]

BINARY

El operador BINARY trata la cadena que le sigue como una cadena binaria. Esta es una forma sencilla de forzar una comparación de cadena en modo case-sensitive incluso si la columna no se ha definido como BINARY o BLOB:

```
mysql> SELECT "a" = "A";
->1

mysql> SELECT BINARY "a" = "A";
->0
```

BINARY string es una forma abreviada de CAST(string AS BINARY). Puedes ver la sección 6.3.5. [Cast functions]. BINARY fue introducida en MySQL versión 3.23.0. Cabe anotar que en ciertos contextos MySQL no estará disponible para utilizar el índice eficientemente cuando conviertas una columna indexada a BINARY.

Si quieres comparar un dato BLOB de modo case-insensitive, siempre lo puedes convertirlo a mayúsculas antes de realizar la comparación.

SELECT ‘ A ’ LIKE UPPER(blob_col) FROM table_name;

Planificamos que en breve la conversión entre diferentes juegos de caracteres para realizar comparaciones de cadenas aún más flexibles.

6.3.3. Funciones numéricas.

6.3.3.1. Operaciones aritméticas

Los operadores aritméticos usuales están disponibles. Cabe anotar que en el caso de ‘- ’, ‘+ ’, y ‘* ’, el resultado se calcula con precisión BIGINT (64 bits) si ambos argumentos son enteros! Si uno de los argumentos es un entero sin signo, y el otro argumento es también un entero, será un entero sin signo. Puedes ver la sección 6.3.5. [Cast functions].

Operador	Descripción
+	Adición: mysql> SELECT 3+5; ->8
-	Substracción: mysql> SELECT 3-5; ->-2
*	Multiplicación: mysql> SELECT 3*5; ->15 mysql> SELECT 18014398509481984*18014398509481984.0; ->324518553658426726783156020576256.0 mysql> SELECT 18014398509481984*18014398509481984; ->0 El resultado de la expresión anterior es incorrecto porque el resultado de la multiplicación entera excede el rango de 64 bits para cálculos BIGINT.
/	División: mysql> SELECT 3/5; ->0.60 La división por cero produce un resultado NULL: mysql> SELECT 102/(1-1); ->NULL Una división se calculará con aritmética BIGINT sólo si se realiza en un contexto en el que el resultado será convertido a entero!

6.3.3.2. Funciones matemáticas

Todas las funciones matemáticas retornan NULL en caso de error.

-

Menos unario. Cambia el signo del argumento:

```
mysql> SELECT – 2;
->-2
```

Cabe anotar que si este operador se utiliza con un BIGINT, el valor de retorno es un BIGINT! Esto significa que deberías evitar el uso de – en enteros que puedan tener el valor de –2^63!

ABS(X)

Retorna el valor absoluto de X:

```
mysql> SELECT ABS(2);
->2

mysql> SELECT ABS(-32);
->32
```

Esta función soporta el uso de valores BIGINT.

SIGN(X)

Retorna el signo del argumento como –1, 0 o 1, dependiendo de si X es negativo, cero, o positivo:

```
mysql> SELECT SIGN(-32);
->-1

mysql> SELECT SIGN(0);
->0

mysql> SELECT SIGN(234);
->1
```

MOD(N,M)
%

Módulo aritmético (como el operador % en C). Retorna el resto de N dividido entre M:

```
mysql> SELECT MOD(234,10);
->4

mysql> SELECT 253 % 7;
->1

mysql> SELECT MOD(29,9);
->2
```

Esta función soporta el uso de valores BIGINT.

FLOOR(X)

Retorna el mayor valor entero no mayor que X:

```
mysql> SELECT FLOOR(1.23);
->1

mysql> SELECT FLOOR(-1.23);
->-2
```

Cabe anotar que el valor de retorno se convierte a BIGINT!

CEILING(X)

Retorna el menor entero no inferior a X:

```
mysql> SELECT CEILING(1.23);
->2

mysql> SELECT CEILING(-1.23);
->-1
```

Cabe anotar que el valor de retorno se convierte a BIGINT!

ROUND(X)

Retorna el argumento X, redondeado al entero más cercano:

```
mysql> SELECT ROUND(-1.23);
->-1

mysql> SELECT ROUND(-1.58);
->-2

mysql> SELECT ROUND(1.58);
->2
```

Cabe anotar que el comportamiento de ROUND() cuando el argumento está a medio camino entre dos enteros depende de la ímplementación de la librería de C. Algunos redondean al número par más cercano, siempre hacia arriba, siempre hacia abajo, o siempre hacia cero. Si necesitas un tipo de redondeo, debes utilizar una función bien definida, como TRUNCATE() o FLOOR() en vez de ROUND.

ROUND(X,D)

Retorna el argumento X, redondeado a un número con D decimales. Si D es 0, el resultado no tendrá punto decimal o parte fraccionaria:

```
mysql> SELECT ROUND(1.298,1);
->1.3

mysql> SELECT ROUND(1.298,0);
->1
```

EXP(X)

Retorna el valor de e (la base de los logaritmos naturales) elevado a la potencia de X:

```
mysql> SELECT EXP(2);
->7.389056

mysql> SELECT EXP(-2);
->0.135335
```

LN(X)

Retorna el logaritmo natural de X:

```
mysql> SELECT LN(2);
->0.693147

mysql> SELECT LN(-2);
->NULL
```

Esta función fue añadida en la versión 4.0.3 de MySQL. Es sinónimo de LOG(X) en MySQL.

LOG(X)
LOG(B,X)

Si se llama con un parámetro, esta función retorna el logaritmo natural de X:

```
mysql> SELECT LOG(2);
->0.693147

mysql> SELECT LOG(-2);
->NULL
```

Si se llama con dos parámetros, esta función retorna el logaritmo de X para una base arbitraria B:

```
mysql> SELECT LOG(2,65536);
->16.000000

mysql> SELECT LOG(1,100);
->NULL
```

La opción de la base arbitraria fue añadida en la versión 4.0.3 de MySQL. LOG(B,X) es equivalente a LOG(X)/LOG(B).

LOG2(X)
Retorna el logaritmo en base 2 de X:

```
mysql> SELECT LOG2(65536);
->16.000000

mysql> SELECT LOG2(-100);
->NULL
```

LOG2() es útil para saber cuántos bits se requieren de almacenamiento para un número. Esta función fue añadida en la versión 4.0.3. de MySQL. En versiones anteriores, puedes utilizar LOG(X)/LOG(2) a cambio.

LOG10(X)

Retorna el logaritmo en base 10 de X.

```
mysql> SELECT LOG10(2);
->0.301030

mysql> SELECT LOG10(100);
->2

mysql> SELECT LOG10(-100);
->NULL
```

POW(X,Y)
POWER(X,Y)

Retorna el valor de X elevado a la Y-ésima potencia:

```
mysql> SELECT POW(2,2);
->4.000000

mysql> SELECT POW(2,-2);
->4.472136
```

PI()

Retorna el valor de PI. El número de decimales visualizado por defecto es 5, pero MySQL utiliza internamente la doble precisión de PI.

```
mysql> SELECT PI();
->3.141593

mysql> SELECT PI()+0.000000000000000000;
->3.141592653589793116
```

COS(X)

Retorna el coseno de X, donde X se da en radianes:

```
mysql> SELECT COS(PI());
->-1.000000
```

SIN(X)

Retorna el seno de X, donde X se da en radianes:

```
mysql> SELECT COS(PI());
->0.000000
```

TAN(X)

Retorna la tangente de X, donde X se da en radianes:

```
mysql> SELECT TAN(PI()+1);
->1.557408
```

ACOS(X)

Retorna el arco del coseno de X, esto es, el valor para el cual el coseno es X. Retorna NULL si X no se halla en el rango de –1 a 1:

```
mysql> SELECT ACOS(1);
->0.000000

mysql> SELECT ACOS(1.0001);
->NULL

mysql> SELECT ACOS(0);
->1.570796
```

ASIN(X)

Retorna el arco del seno de X, esto es, el valor para el cual el seno es X. Retorna NULL si X no se halla en el rango de –1 a 1.

```
mysql> SELECT ASIN(0.2);
->0.201358

mysql> SELECT ASIN('foo');
->0.000000
```

ATAN(X)

Retorna el arco de la tangente de X, esto es, el valor para el cual la tangente es X:

```
mysql> SELECT ATAN(2);
->1.107149

mysql> SELECT ATAN(-2);
->-1.107149
```

ATAN(Y,X)
ATAN2(Y,X)

Retorna el arco tangente de las dos variables X e Y. Esto es similar a calcular el arco tangente de Y/X, excepto en que los signos de los dos argumentos se utilizan para determinar el cuadrante del resultado:

```
mysql> SELECT ATAN(-2,2);
->-0.785398

mysql> SELECT ATAN2(PI(),0);
```

->1.570796

COT(X)

Retorna la cotangente de X:

mysql> SELECT COT(12);
->-1.57267341

mysql> SELECT COT(0);
->NULL

RAND()
RAND(N)

Retorna un valor de coma flotante aleatorio dentro del rango 0 a 1.0. Si se especifica un valor entero N, se utiliza como semilla:

mysql> SELECT RAND();
->0.9233482386203

mysql> SELECT RAND(20);
->0.15888261251047

mysql> SELECT RAND();
->0.63553050033332

mysql> SELECT RAND();
->0.70100469486881

No puedes utilizar una columna con valores de RAND() en una cláusula ORDER BY, debido a que ORDER BY evaluaría la columna múltiples ocasiones. En MySQL 3.23, puedes, sin embargo, hacer: SELECT * FROM table_name ORDER BY RAND().

Esto es útil para conseguir una muestra aleatoria de un conjunto SELECT * FROM table1, table2 WHERE a=b AND c<d ORDER BY RAND() LIMIT 1000.

Cabe anotar que un RAND() en una cláusula WHERE se re-evaluará cada vez que se ejecute WHERE.

No es que RAND() sea un generador de números aleatorios perfecto, pero sí que es una forma rápida de generar ad hoc valores aleatorios, portable entre plataformas por la misma versión de MySQL.

LEAST(X,Y,...)

Con dos o más argumentos, retorna el argumento más pequeño (menos valorado). Los argumentos se comparan utilizando las siguientes reglas:

- Si el valor de retorno se utiliza en un contexto INTEGER, o todos los argumentos son enteros, se compararán como enteros.
- Si el valor de retorno se usará en un contexto REAL, o todos los argumentos son valores reales, serán comparados como valores reales.

- Si cualquier argumento es una cadena case-sensitive, los argumentos se compararán en modo case-sensitive.
- En otros casos, los argumentos se compararán como cadenas case-insensitive:

```
mysql> SELECT LEAST(2,0);
->0

mysql> SELECT LEAST(34.0,3.0,5.0,767.0);
->3.0

mysql> SELECT LEAST("B","A","C");
->"A"
```

En versiones de MySQL anteriores a 3.22.5 puedes utilizar MIN() en vez de LEAST.

GREATEST(X,Y,...)

Retorna el argumento de mayor valor (de máximo valor). Los argumentos se comparan utilizando las mismas reglas que para LEAST:

```
mysql> SELECT GREATEST(2,0);
->2

mysql> SELECT GREATEST(34.0,3.0,5.0,767.0);
->767.0

mysql> SELECT GREATEST("B","A","C");
->"C"
```

En versiones de MySQL anteriores a 3.22.5 puedes utilizar MAX() en vez de GREATEST.

DEGREES(X)

Retorna el argumento X, convertido de radianes a grados.

```
mysql> SELECT DEGREES(PI())
->180.000000
```

RADIANS(X)

Retorna el argumento X, convertido de grados a radianes.

```
mysql> SELECT RADIANS(90)
->1.570796
```

TRUNCATE(X,D)

Retorna el número X, truncado a D decimales. Si D es 0, el resultado no tendrá punto decimal o parte fraccionaria:

```
mysql> SELECT TRUNCATE(1.223,1)
->1.2
```

```
mysql> SELECT TRUNCATE(1.999,1)
->1.9

mysql> SELECT TRUNCATE(1.999,0)
->1

mysql> SELECT TRUNCATE(-1.999,1)
->-1.9
```

A partir de MySQL 3.23.51 todos los números son redondeados a cero. Si D es negativo, la totalidad del número es convertido a cero:

```
mysql> SELECT TRUNCATE(122,-2)
->100
```

Cabe anotar que del mismo modo que decimales no son almacenados normalmente como valores exactos en los ordenadores, sino como valores de precisión doble, puedes encontrarte con resultados como el siguiente:

```
mysql> SELECT TRUNCATE(10.28*100,0)
->1027
```

Lo anterior sucede porque 10.28 se almacena como algo parecido a 10.27999999999999.

6.3.4. Funciones de fecha y hora

Puedes ver la sección 6.2.2. para una descripción del rango de valores de cada tipo tiene y los formatos válidos de los valores en los que la fecha y hora pueden especificarse.

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

```
mysql> SELECT something FROM tbl_name WHERE TO_DAYS(NOW()) – TO_DAYS
(date_col) <= 30;
```

DAYOFWEEK(date)

Retorna el índice del día de semana para la fecha dada (1 = domingo, 2 = lunes, ... 7= sábado). Estos valores índice se corresponden con el estándar ODBC.

```
mysql> SELECT DAYOFWEEK('1998-02-03');
->3
```

WEEKDAY(date)

Retorna el índice del día de la semana para la fecha (0=lunes, 1=martes,..., 6 = domingo):

```
mysql> SELECT WEEKDAY('1997-10-04 22:23:00');
->5

mysql> SELECT WEEKDAY('1997-11-05');
->2
```

DAYOFMONTH(date)

Retorna el día del mes para la fecha, dentro del rango 1 a 31:

```
mysql> SELECT DAYOFMONTH('1998-02-03');
->3
```

DAYOFYEAR(date)

Retorna el día del año para la fecha, en el rango de 1 a 366:

```
mysql> SELECT DAYOFYEAR('1998-02-03');
->34
```

MONTH(date)

Retorna el mes de la fecha, en el rango de 1 a 12:

```
mysql> SELECT MONTH('1998-02-03');
->2
```

DAYNAME(date)

Retorna el nombre del día de la semana de la fecha:

```
mysql> SELECT DAYNAME("1998-02-03");
->'Thursday'
```

MONTHNAME(date)

Retorna el nombre del mes de la fecha:

```
mysql> SELECT MONTHNAME("1998-02-05");
->'February'
```

QUARTER(date)

Retorna el trimestre de la fecha, en el rango de 1 a 4:

```
mysql> SELECT QUARTER('98-04-01');
->2
```

WEEK(date)
WEEK(date,first)

Con un solo argumento, retorna la semana de la fecha, en el rango de 0 a 53 (sy, puede ser que sean los inicios de la semana 53), para localidades en los que el domingo sea el primer día de la semana. La forma en dos argumentos de WEEK() te permite especificar si la semana empieza en domingo o en lunes. La semana empieza en domingo si el segundo argumento es 0, y en lunes si el segundo argumento es 1:

```
mysql> SELECT WEEK('1998-02-20');
->7
```

```
mysql> SELECT WEEK('1998-02-20',0);
->7

mysql> SELECT WEEK('1998-02-20',1);
->8

mysql> SELECT WEEK('1998-12-31',1);
->53
```

Cabe anotar que en la versión 4.0, WEEK(#,0) cambió para ajustarse al calendario de los Estados Unidos.

YEAR(date)

Retorna el año de la fecha, en el rango de 1000 a 9999:

```
mysql> SELECT YEAR('98-02-03');
->1998
```

YEARWEEK(date)
YEARWEEK(date,first)

Retorna el año y la semana de la fecha. El segundo argumento funciona exactamente igual que el segundo argumento de WEEK(). Cabe anotar que el año puede variar de año para la primera y última semana del año:

```
mysql> SELECT YEARWEEK('1987-01-01');
->198653
```

HOUR(time)

Retorna la hora para el dato dado, en el rango de 0 a 23:

```
mysql> SELECT HOUR('10:05:03');
->10
```

MINUTE(time)

Retorna los minutos del dato dado, en el rango de 0 a 59:

```
mysql> SELECT HOUR('98-02-03 10:05:03');
->5
```

SECOND(time)

Retorna los segundos para el dato dado, en el rango de 0 a 59:

```
mysql> SELECT SECOND('10:05:03');
->3
```

PERIOD_ADD(P,N)

Añade N meses al período P (en el formato YYMM o YYYYMM). Retorna un valor en el formato YYYYMM.

Nota que el argumento de período P no es un valor de fecha:

mysql> SELECT PERIOD_ADD(9801,2);
->199803

PERIOD_DIFF(P1,P2)

Retorna el número de meses entre los períodos P1 y P2. P1 y P2 deberían estar en el formato YYMM o YYYYMM.

Nota que los argumentos de período P1 y P2 no son valores de fecha:

mysql> SELECT PERIOD_DIFF(9802,199703);
->11

DATE_ADD(date,INTERVAL expr type)
DATE_SUB(date,INTERVAL expr type)
ADDDATE(date,INTERVAL expr type)
SUBDATE(date,INTERVAL expr type)

Estas funciones realizan la aritmética de fechas. Se incorporaron en la versión 3.22 de MySQL. ADDDATE() y SUBDATE() son sinónimos para DATE_ADD() y DATE_SUB().

En la versión 3.23 de MySQL, puedes utilizar + y – en vez de DATE_ADD() y DATE_SUB() si la expresión a la derecha es una columna date o datetime (puedes ver el ejemplo posterior).

date es un valor DATETIME o DATE especificando la fecha de partida. expr es una expresión especificando el valor de intervalo a ser añadido o substraído de la fecha de partida. expr es una cadena; puede empezar con un '-' para intervalos negativos. type es una palabra clave que indica como se debería interpretar la expresión.

La función relacionada EXTRACT(type FROM date) retorna el intervalo 'type' desde la fecha.

La tabla siguiente visualiza cómo los argumentos de tipo y expr se relacionan:

Tipo	Formato esperado para expr
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
MONTH	MONTHS
YEAR	YEARS
MINUTE_SECOND	" MINUTES:SECONDS"
HOUR_MINUTE	" HOURS:MINUTES"
DAY_HOUR	" DAYS HOURS"
YEAR_MONTH	" YEARS-MONTHS"
HOUR_SECOND	" HOURS:MINUTES:SECONDS"
DAY_MINUTE	" DAYS HOURS:MINUTES"
DAY_SECOND	" DAYS HOURS:MINUTES:SECONDS"

MySQL permite cualquier delimitador de puntuación en el formato de expr. Los expuestos en la tabla simplemente son delimitadores sugeridos. Si el argumento date es un valor DATE y tus cálculos incluyen sólo las partes YEAR, MONTH y DAY (es decir, sin horas), el resultado es un valor DATE. En el resto de casos, el resultado es un valor DATETIME:

```
mysql> SELECT "1997-12-31 23:59:59" + INTERVAL 1 SECOND;
->1998-01-01 00:00:00

mysql> SELECT INTERVAL 1 DAY + "1997-12-31";
->1998-01-01

mysql> SELECT "1998-01-01" + INTERVAL 1 SECOND;
->1997-01-01 23:59:59

mysql> SELECT DATE_ADD("1997-12-31 23:59:59", INTERVAL 1 SECOND);
->1998-01-01 00:00:00

mysql> SELECT DATE_ADD("1997-12-31 23:59:59", INTERVAL "1:1" MINUTE_SECOND);
->1998-01-01 22:58:59

mysql> SELECT DATE_ADD("1997-12-31 23:59:59", INTERVAL "1 1:1:1" DAY_SECOND);
->1997-12-30 22:58:59

mysql> SELECT DATE_ADD("1998-01-01 00:00:00", INTERVAL "- 1 10" DAY_HOUR);
->1997-12-30 14:00:00

mysql> SELECT DATE_SUB("1998-01-02", INTERVAL 31 DAY);
->1997-12-02
```

Si especificas un valor interno demasiado corto (no incluye todas las partes del intervalo que se esperarían de la palabra clave type), MySQL asume que has dejado las partes más a la izquierda del valor del intervalo. Por ejemplo, si especificas un type de DAY_SECOND, el valor de expr se espera que sean días, horas, minutos y segundos. Si especificas un valor como "1:10", MySQL asume que las partes de los días y las horasno se hallan y el valor representa minutos y segundos. En otras palabras, "1:10" DAY_SECOND se interpretan de modo equivalente a "1:10" MINUTE_SECOND. Esto es análogo al modo en el que MySQL interpreta los valores TIME que represente el tiempo transcurrido más que la hora del día.

Nota que si añades o substraes un valor de fecha de algo que contenga una parte horaria, el valor se convertirá automáticamente a DATETIME:

```
mysql> SELECT DATE_ADD("1999-01-01", INTERVAL 1 DAY);
->1991-01-02

mysql> SELECT DATE_ADD("1999-01-01", INTERVAL 1 HOUR);
->1991-01-01 01:00:00
```

Si utilizas fechas realmente incorrectas, el resultado será NULL. Si añades MONTH, YEAR_MONTH, o YEAR y la fecha resultante tiene un día superior al máximo día para el nuevo mes, el día se ajusta a los días máximos del nuevo mes:

```
mysql> SELECT DATE_ADD("1998-01-30", INTERVAL 1 MONTH);
->1991-02-28
```

Nota que a partir del ejemplo precedente la palabra INTERVAL y la palabra clave type no son case-sensitive.

EXTRACT(type FROM date)

La función EXTRACT() utiliza los mismos tipos de tipos de especificadores de intervalos como DATE_ADD() o DATE_SUB(), pero extrae partes de la fecha, más que realizar aritmética de fechas.

```
mysql> SELECT EXTRACT(YEAR FROM “1999-07-02”);
->1999

mysql> SELECT EXTRACT(YEAR_MONTH FROM “1999-07-02 01:02:03”);
->199907

mysql> SELECT EXTRACT(DAY_MINUTE FROM “1999-07-02 01:02:03”);
->20102
```

TO_DAYS(date)

Dada una fecha date, retorna el número de día (el número de día desde el año 0):

```
mysql> SELECT TO_DAYS(950501);
->728779

mysql> SELECT TO_DAYS(‘1997-10-07’);
->729669
```

TO_DAYS() no se ha desarrollado para ser utilizado en fechas anteriores al advenimiento del calendario gregoriano (1582), debido a que no tiene en cuenta los días perdidos en el momento del cambio de calendario.

DATE_FORMAT(date,format)

Formatea la fecha date de acuerdo con la cadena format. Los siguientes especificadores pueden ser utilizados en la cadena format:

Especificador	Descripción
%M	Nombre del mes (January...December)
%W	Nombre del día de la semana (Sunday...Saturday)
%D	Día de la semana con sufijo inglés (1st, 2nd, 3rd...)
%Y	Año, numérico, 4 dígitos
%y	Año, numérico, 2 dígitos
%X	Año para la semana para la que Sunday es el primer día de la semana, numérico, 4 dígitos, utilizado con ‘%V’
%x	Año para la semana para la que Monday es el primer día de la semana, numérico, 4 dígitos, utilizado con ‘%v’
%a	Día abreviado de la semana (Sun...Sat)
%d	Día del mes, numérico (00..31)
%e	Día del mes, numérico (0..31)
%m	Mes, numérico (01..12)
%c	Mes, numérico (1..12)

%b	Forma abreviada del nombre de mes (Jan..Dec)
%j	Día del año (001..366)
%H	Hora (00..23)
%k	Hora (0..23)
%h	Hora (01..12)
%I	Hora (01..12)
%l	Hora (1..12)
%i	Minutos, numérico (00..59)
%r	Hora, formato 12 horas (hh:mm:ss [AP]M)
%T	Hora, 24 horas (hh:mm:ss)
%S	segundos (00..59)
%s	segundos (00..59)
%p	AM o PM
%w	Día de la semana (0 = Sunday... 6=Saturday)
%U	Semana (00..53), donde Sunday es el primer día de la semana
%u	Semana (00..53), donde Monday es el primer día de la semana
%V	Semana (01..53), donde Sunday es el primer día de la semana. Utilizado con '%X'
%v	Semana (01..53), donde Monday es el primer día de la semana. Utilizado con '%x'
%%	' %' literal

El resto de caracteres son copiados del formato al resultado sin interpretación:

```
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');
->'4 th 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'
```

En la versión 3.23 dfe MySQL y posteriores el carácter "%" se requiere antes del carácter especificador de formato, en versiones anteriores de MySQL, '%' era opcional.

TIME_FORMAT(time,format)

Se utiliza como DATE_FORMAT(), pero la cadena format sólo puede contener aquellos especificadores de formato que manejen horas, minutos y segundos. El resto de especificadores producen un valor NULL o 0.

CURDATE()
CURRENT_DATE

Retorna la fecha de hoy como un valor con formato 'YYYY-MM-DD' o YYYYMMDD, dependiendo del contexto (cadena o numérico) en el que se está utilizando la función:

```
mysql> SELECT CURDATE();
->'1997-12-15'
mysql> SELECT CURDATE()+0;
->19971215
```

CURTIME()
CURRENT_TIME

Retorna la hora actual como un valor en formato 'HH:MM:SS' o HHMMSS, dependiendo del contexto (cadena o numérico) en el que se está utilizando la función:

```
mysql> SELECT CURTIME();
->'23:50:26'

mysql> SELECT CURTIME()+0;
->235026
```

NOW()
SYSDATE()
CURRENT_TIMESTAMP

Retorna la fecha actual y la hora como un valor en formato 'YYYY-MM-DD HH:MM:SS' o YYYYMMDDHHMMSS, dependiendo del contexto (cadena o numérico) en el que se está utilizando la función:

```
mysql> SELECT NOW();
->'1997-12-15 23:50:26'

mysql> SELECT NOW()+0;
->19971215235026
```

Nota que NOW() sólo se evalúa una vez por consulta, al principio de la ejecución de la consulta. Esto significa que las múltiples referencias de NOW() dentro de una misma consulta siempre darán el mismo valor.

UNIX_TIMESTAMP()
UNIX_TIMESTAMP(date)

Si se llama sin argumentos, retorna un timestamp en formato UNIX (segundos desde '1970-01-01 00:00:00 GMT) como entero sin signo. Si UNIX_TIMESTAMP() se llama con el argumento date, retorna el valor del argumento como segundos desde '1970-01-01 00:00:00' GMT. El contenido de date puede ser una cadena DATE, una cadena DATETIME, una TIMESTAMP, o un número en formato YYMMDD o YYYYMMDD en hora local:

```
mysql> SELECT TIMESTAMP();
->882226357

mysql> SELECT UNIX_TIMESTAMP('1997-10-04 22:23:00');
->875996580
```

Cuando UNIX_TIMESTAMP se utiliza en una columna TIMESTAMP, la función retornará el valor interno timestamp directamente, sin conversión implícita de cadena a timestamp. Si pasas

una fecha fuera de rango a UNIX_TIMESTAMP() retornará 0, pero nota que sólo se realiza un control básico (año entre 1970 y 2037, mes entre 1 y 12, día entre 1 y 31).

Si quieres realizar una substracción entre dos columnas UNIX_TIMESTAMP(), puedes querer convertir el resultado a enteros con signo. Puedes ver la sección 6.3.5 [Cast functions].

FROM_UNIXTIME(unix_timestamp)

Retorna una representación del argumento unix_timestamp como un valor en formato 'YYYY-MM-DD HH:MM:SS' o YYYYMMDDHHMMSS, dependiendo del contexto en el que se utilice esta función:

```
mysql> SELECT FROM_UNIXTIME(875996580);
->'1997-10-04 22:23:00'

mysql> SELECT FROM_UNIXTIME(875996580)+0;
->'19971004222300'
```

FROM_UNIXTIME(unix_timestamp,format)

Retorna una representación del argumento unix_timestamp formateado de acuerdo con la cadena format. Format puede contener los mismos especificadores listados anteriormente en la función DATE_FORMAT():

```
mysql> SELECT FROM_UNIXTIME(UNIX_TIMESTAMP(),'%Y %D %M %h:%i:%s %x');
->'1997 23rd December 03:43:30 1997'
```

SEC_TO_TIME(seconds)

Retorna el argumento seconds, convertido a horas, minutos y segundos, como un valor en formato 'HH:MM:SS' o HHMMSS, dependiendo de si el contexto en el que se utiliza la función es numérico o de cadena:

```
mysql> SELECT SEC_TO_TIME(2378);
->'00:39:38'

mysql> SELECT SEC_TO_TIME(2378)+0;
->3938
```

TIME_TO_SEC(time)

Retorna el argumento time, convertido en segundos:

```
mysql> SELECT TIME_TO_SEC('22:23:00');
->80580

mysql> SELECT TIME_TO_SEC('00:39:38');
->2378
```

6.3.5. Funciones de cambios de tipo [Cast functions]

La sintaxis de la función CAST es:

CAST(expression AS type)

o:

CONVERT(expression,type)

Donde type es uno de los tipos siguientes:

- BINARY.
- DATE.
- DATETIME.
- SIGNED{INTEGER}
- TIME.
- UNSIGNED{INTEGER}

CAST() se basa en la sintaxis ANSI SQL 92 y CONVERT() se basa en la sintaxis ODBC. La función de cambio de tipos es útil principalmente cuando quieres crear una columna con un tipo específico en un sentencia CREATE...SELECT:

CREATE new_table SELECT CAST('2000-01-01' AS DATE);

CAST(string AS BINARY) es la mismo que BINARY string.

Para convertir una cadena a un valor numérico, normalmente no necesitas hacer nada; sólo utiliza el valor de la cadena como si fuera un número:

```
mysql> SELECT 1+'1';
->2
```

MySQL soporta la aritmética en valores con y sin signo de 64 bits. Si estás utilizando operadores numéricos (como +) y uno de los operandos son unsigned integer, el resultado será unsigned. Puedes obviarlo usando los operadores de conversión SIGNED y UNSIGNED, que convertirá la operación a un entero con o sin signo de 64 bits, respectivamente:

```
mysql> SELECT CAST(1-2 AS UNSIGNED);
->18446744073709551615

mysql> SELECT CAST((1-2 AS UNSIGNED) AS SIGNED);
->-1
```

Nota que si la operación es un valor de coma flotante (DECIMAL() en este contexto es considerado como un valor de coma flotante) el resultado será un valor de coma flotante que no se ve afectado por la regla anterior.

```
mysql> SELECT CAST(1 AS UNSIGNED)-2.0;
->-1.0
```

Si estás utilizando una cadena en una operación aritmética, ésta se convierte a un valor de coma flotante.

Las funciones de CAST() y CONVERT() se añadieron en MySQL 4.0.2.

El manejo de valores sin signo fue cambiada en MySQL 4.0 para soportar los valores BIGINT propiamente. Si tienes código en el que quieras utilizar simultáneamente MySQL 4.0 y 3.23 (y en tal caso probablemente no podrás utilizar la función CAST), puedes utilizar el siguiente truco para conseguir un resultado con signo al substraer dos columnas enteras sin signo:

mysql> SELECT (unsigned_column_1+0.0)-(unsigned_column_2+0.0);

La idea es que las columnas se convierten a valores de coma flotante antes de realizar la substracción.

Si tienes un problema con columnas unsigned en tu vieja aplicación para MySQL al portarla a MySQL 4.0, puedes utilizar la opción --sql-mode=NO_UNSIGNED_SUBSTRACTION al iniciar mysqld. Nota sin embargo que a medida que lo uses, no podrás realizar un uso eficiente del tipo de columna UNSIGNED BIGINT.

6.3.6. Otras funciones

6.3.6.1. Funciones con bits

MySQL utiliza la aritmética de BIGINT (64 bits) para las operaciones con bits, por lo que estas operaciones tienen un rango máximo de 64 bits.

| OR bit a bit [bitwise]

mysql> SELECT 29 | 15;
->31

El resultado es un entero sin signo de 64 bits.

& AND bit a bit

mysql> SELECT 29 & 15;
->13

El resultado es un entero sin signo de 64 bits.

^ XOR bit a bit

mysql> SELECT 1^1;
->0

mysql> SELECT 1^0;
->1

mysql> SELECT 11^3;
->8

El resultado es un entero sin signo de 64 bits.

<<

Desplaza los bits hacia la izquierda

```
mysql> SELECT 1<<2;
->4
```

El resultado es un entero sin signo de 64 bits.

>>

desplaza los bits hacia la derecha

```
mysql> SELECT 4>>2;
->1
```

El resultado es un entero sin signo de 64 bits.

~

Invierte todos los bits

```
mysql> SELECT 5 & ~1;
->4
```

El resultado es un entero sin signo de 64 bits.

BIT_COUNT(N)

Retorna el número de bits marcados en el argumento N:

```
mysql> SELECT BIT_COUNT(29);
->4
```

6.3.6.2. Funciones misceláneas

DATABASE()

Retorna el nombre de la base de datos actual:

```
mysql> SELECT DATABASE();
->'test'
```

Si no hay una base de datos activa, DATABASE() retorna una cadena vacía.

USER()
SYSTEM_USER()
SESSION_USER()

Retorna el usuario actual de MySQL:

```
mysql>SELECT USER();
->davida@localhost
```

En la versión 3.22.11 de MySQL o posterior, incluye el nombre de host del cliente, además del nombre de usuario. Puedes extraer sólo la parte del nombre de usuario así (que funciona donde el valor incluya la parte de hostname):

```
mysql>SELECT SUBSTRING_INDEX(USER(),"@",1);
->davida
```

PASSWORD(str)

Calcula la cadena de password a partir del password str en texto plano. Esta es la función utilizada para encriptar los passwords de MySQL al almacenarlos en la columna Password de la tabla user:

```
mysql>SELECT PASSWORD('badpwd');
->'7f84554057dd964b'
```

La encriptación de PASSWORD() es irreversible.

PASSWORD() no realiza la encriptación de la contraseña del mismo modo que lo hacen los passwords en Unix. No deberías asumir que si tus contraseñas de MySQL y Unix son las mismas, el valo encriptado será el mismo en ambos. Puedes ver ENCRYPT().

ENCRYPT(str[,salt])

Encripta str utilizando la llamada a crypt() de Unix. El argumento salt debería ser una cadena con dos caracteres. (En la versión 3.22.16 de MySQL, salt puede tener más de dos caracteres):

```
mysql>SELECT ENCRYPT("hello");
->'VxuFAJXVARROc'
```

Si crypt() no está disponible en tu sistema, ENCRYPT() siempre retorna NULL.
ENCRYPT() ignora más a llá de los primeros 8 caracteres de str, al menos en algunos sistemas. Esto vendrá determinado por el comportamiento de la llamada subyacente al sistema de crypt().

ENCODE(str,pass_str)

Encripta str utilizando pass_str como contraseña. Para desencriptar el resultado, utiliza DECODE().

El resultado es una cadena binaria de la misma longitud que string. Si quieres guardarlo en una columna, utiliza una columna de tipo BLOB.

DECODE(crypt_str,pass_str)

Desencripta la cadena encriptada crypt_str utilizando pass_str como password. crypt_str debería ser una cadena retornada de ENCODE().

MD5(string)

Calculas una checksum de 128 bits MD5 para la cadena. El valor es retornado como un valor hexadecimal de 32 dígitos que puede, por ejemplo, ser utilizado como clave de hash:

```
mysql>SELECT MD5("testing")
->'ae2b1fca515949e5d54fb22b8ed95575'
```

Este es el algoritmo “RSA Data Security, Inc. MD5 Message-Digest”.

SHA1(string)
SHA(string)

Calculas una checksum de 160 bits SHA1 para la cadena, como se describee en la RFC 3174 (Secure Hash Algorithm). El valor es retornado como un número hexadecima de 40 dígitos, o NULL en caso del que el argumento de entrada sea NULL. Una de las posibles utilidades para esta función és como generador de claves de hash. También puedes utilizarlo como una función criptigráficamente segura para almacenar contraseñas.

mysql>SELECT SHA1(“abc”);
->‘a9993e364706816aba3e2571850c26c9cd0d89d’

SHA1() se añadió en la versión 4.0.2. y puede ser considerada un equivalente criptigráfico más seguro que MD5(). SHA() es sinónimo de SHA1().

AES_ENCRYPT(string,key_string)
AES_DECRYPT(string,key_string)

Estas funciones permiten la encriptación y desencriptación de los datos utilizando el algoritmo oficial AES (Advanced Encryption Standard), conocido anteriormente como Rijndael. Se utiliza la codificación de 128 bits, pero puedes extenderlo hasta 256 bits parcheando el código fuente. Nosotros hemos escogido 128 bits porque es más rápido y habitualmente es lo suficientemente seguro.

Los argumentos de entrada pueden ser de cualquier longitud. Si el argumento es NULL, el resultado de esta función también es NULL.

Ya que AES es un algoritmo de encriptación de bloques, se utiliza el relleno para cadenas de longitud impar, y así la longitud resultante de la cadena puede ser calculada como 16*(trunc (string_length/16)+1).

Si AES_DECRYPT() detecta datos inválidos o relleno incorrecto, retorna NULL.
Sin embargo, es posible que AES_DECRYPT() retorne un valor no nulo (posiblemente basura) si los datos de entrada o la clave fueran inválidos.

Puedes utilizar las funciones AES para almacenar datos en forma encriptada al modificar tus consultas:

mysql>INSERT INTO t VALUES (1,AES_ENCRYPT(“text”,”password”));

Puedes conseguir aún mas seguridad evitando de transferir las claves en las conexiones de cada consulta, objetivo conseguible almacenando en el lado de servidor una variable en tiempo de conexión:

SELECT @password:=”my password”;
INSERT INTO t VALUES (1,AES_ENCRYPT(“text,@password”));

AES_ENCRYPT() y AES_DECRYPT() fueron añadidas en la versión 4.0.2, y pueden ser consideradas las criptográficamente más seguras disponibles actualmente en MySQL.

DES_ENCRYPT(string_to_encrypt[, (key_number|key_string)])

Encripta la cadena con la clave dada utilizando el algoritmo DES.
Nota que esta función sólo trabaja si tienes configurada MySQL con soporte SSL. Puedes ver la sección 4.3.9 [Secure connections].

La clave de encriptación a usar es seleccionada del siguiente modo:

Argumento	Descripción
Sólo un argumento	Se utiliza la primera clave de des-key-file
Clave numérica	Se utiliza la clave dada (0-9) de des-key-file
Cadena	La clave key_string dada se utilizará para encriptar string_to_encrypt.

La cadena de retorno será una cadena binaria donde el primer carácter será CHAR(128| key_number).

El 128 se añade para reconocer más fácilmente una clave encriptada. Si utilizas una clave de cadena, key_number será 127.
En caso de error, esta función retorna NULL.
La longitud de la cadena para el resultado será new_length=org_length+(8-(org_length % 8))+1.

des-key-file tiene el formato siguiente:

key-number des-key-string
key-number des-key-string

Cada key-number debe tener un número en el rango de 0 a 9. Las líneas de la fila pueden estar en cualquier orden. des-key-string es la cadena que se utilizará para encriptar el mensaje. Entre el número y la clave debería haber al menos un espacio. La primera clave es la clave por defecto que será utilizada si no especificas cualquier argumento clave a DES_ENCRYPT().

Puedes dar a MySQL a leer los nuevos valores claves del archivo de claves con el comando FLUSH DES_KEY_FILE. Esto requiere del privilegio Reload_priv.

Un beneficio de tener un conjunto de claves por defecto es que esta da a las aplicaciones una forma de comprobar la existencia de valores de columnas encriptados, sin dar al usuario final el derecho de desencriptar tales valores.

mysql>SELECT customer_address FROM customer_table WHERE crypted_credit_card=DES_ENCRYPT(“credit_card_number”);

DES_DECRYPT(string_to_decrypt[, key_string])

Desencripta una cadena encriptada con DES_ENCRYPT()

Nota que esta función sólo trabaja si tienes configurada MySQL con soporte SSL. Puedes ver la sección 4.3.9 [Secure connections].

Si no se da ningún argumento key_string, DES_DECRYPT() examina el primer byte de la cadena encriptada para determinar el número clave DES que fue usado para encriptar la cadena original, entonces lee la clave desde des-key-file para desencriptar el mensaje. Para que esto trabaje, el usuario debe disponer del privilegio SUPER.

Si pasas a esta función un argumento des-key-file, esta cadena se utilizará como la clave para desencriptar el mensaje.

Si string_to_decrypt no parece una cadena encriptada, MySQL retornará la dada string_to_decrypt. En caso de error, esta función retorna NULL.

LAST_INSERT_ID([expr])

Retorna el último valor generado automáticamente en una columna AUTO_INCREMENT. Puedes ver la sección 8.4.3.30 [mysql_insert_id()].

```
mysql>SELECT LAST_INSERT_ID();
->195
```

La última ID generada se mantiene en el servidor en una base per-conexión (?). No se cambiará por otro cliente. Incluso no será variado si actualizas la columna AUTO_INCREMENT con un valor non-magic (esto es, un valor que no es ni NULL ni 0).

Si insertas varias filas al mismo tiempo con una sentencia INSERT, LAST_INSERT_ID() retorna el valor de la primera fila insertada. La razón de este hecho es que esto te permite reproducir la misma sentencia contra algún otro servidor.

Si expr viene dado como argumento para LAST_INSERT_ID(), entonces el valor del argumento se retorna por la función, y se sitúa como el siguiente valor a ser retornado por LAST_INSERT_ID(). Esto puede utilizarse para simular secuencias.

Primero crea la tabla:

```
mysql>CREATE TABLE sequence (id INT NOT NULL);
mysql> INSERT INTO sequence VALUES (0);
```

Entonces la tabla puede ser usada para generara secuencias de valores como esta:

```
mysql>UPDATE sequence SET id=LAST_INSERT_ID(id+1);
```

Puedes generar secuencias sin llamar LAST_INSERT_ID(), pero la utilidad de utilizar la función de este modo es que el valor ID se mantiene en el servidor como el último valor generado automáticamente (soporte multiusuario). Puedes recuperar la nueva ID com si leyeras cualquier valor AUTO_INCREMENT normal en MySQL. Por ejemplo, LAST_INSERT_ID() (sin argumento) retornará la nueva ID. La función mysql_insert_id() de la API de C también puede ser usada para conseguir el valor. Nota que dado que mysql_insert_id() sólo se actualiza después de una sentencia INSERT o UPDATE, no puedes utilizar la función de la API de C para recuperar el valor de LAST_INSERT_ID(expr) después de ejecutar otra sentencia SQL, como SELECT o SET.

FORMAT(X,D)

Formatea el número X a un formato del tipo '#,###,###.##', redondeado a D decimales. Si D es 0, el resultado no tendrá punto decimal o parte fraccionaria:

```
mysql>SELECT FORMAT(12332.123456,4);
->'12,332.1235'
```

```
mysql>SELECT FORMAT(12332.1,4);
->'12,332.1000'

mysql>SELECT FORMAT(12332.2,0);
->'12,332'
```

VERSION()

Retorna una cadena indicando la versión del servidor de MySQL:

```
mysql>SELECT VERSION();
->'3.23.13-log'
```

Nota que si tu versión finaliza con –log, significa que el logging está activado.

CONNECTION_ID()

Retorna el id de conexión (thread_id) para la conexión. Cada conexión tiene su propia y única id.

```
mysql>SELECT CONNECTION_ID();
->1
```

GET_LOCK(str,timeout)

Trata de obtener un bloqueo con un nombre dado por la cadena str, con un tiempo de exceso dado por timeout. Retorna 1 si se ha conseguido el bloqueo y 0 en caso contrario, o NULL si ha ocurrido un error (tal como falta de memoria o que el hilo [thread] fue eliminado con mysqladmin kill). Un bloqueo es liberado cuando ejecutas RELEASE_LOCK(), ejecutas un GET_LOCK(), o el hilo termina. Esta función puede ser utilizada para implementar el bloqueo de la aplicación o para simular los bloqueos de registros. Bloquea las demandas de otros clientes por bloqueos con el mismo nombre; los clientes que se ponen de acuerdo en una cadena de bloqueo determinada, pueden utilizar tal cadena para realizar el bloqueo consultivo cooperativo [cooperative advisory locking].

```
mysql>SELECT GET_LOCK("lock1",10);
->1

mysql>SELECT IS_FREE_LOCK("lock2");
->1

mysql>SELECT GET_LOCK("lock2",10);
->1

mysql>SELECT RELEASE_LOCK("lock2");
->1

mysql>SELECT RELEASE_LOCK("lock1");
->NULL
```

Nota que la segunda llamada de segundo RELEASE_LOCK() retorna NULL porque el bloqueo "lock1" fue liberado automáticamente per la segunda llamada GET_LOCK().

RELEASE_LOCK(str)

Libera el bloqueo nombrado por la cadena str que fue obtenida con GET_LOCK(). Retorna 1 si el bloqueo fue liberado, 0 si no estaba bloqueado por este hilo (y en tal caso el bloqueo no se libera), y NULL si el bloqueo citado no existiera. El bloque no existirá si nunca fue obtenido por una llamada a GET_LOCK() o si ha sido ya liberado.

La sentencia DO es la conveniente para ser utilizada con RELEASE_LOCK(). Puedes ver la sección 6.4.10.

IS_FREE_LOCK(str)

Comprueba si el bloqueo llamado str es libre para el uso (esto es, no bloqueado). Retorna 1 si el bloqueo está libre (nadie lo está utilizando), 0 si el bloqueo está en uso, y NULL en caso de errores (como argumentos incorrectos).

BENCHMARK(count,expr)

La función BENCHMARK() ejecuta la expresión expr repetidamente en count ocasiones. Esto puede ser usado tan rápido como MySQL procese la expresión. El valor del resultado es siempre 0. El uso está en el cliente MySQL, que reporta ls tiempos de ejecución de la consulta:

```
mysql>SELECT BENCHMARK(1000000,ENCODE("hello","goodbye"));
+----------------------------------------------------------------+
| BENCHMARK(1000000,ENCODE("hello","goodbye"))           |
+----------------------------------------------------------------+
|                                                                0|
+----------------------------------------------------------------+
1 row in set (4.74 sec)
```

El tiempo reportado es el transcurrido en el destino de cliente, no el tiempo de CPU en el lado de servidor. Puede ser prudente ejecutar BENCHMARK() varias veces, e interpretar el resultado reparando en el nivel de carga de la máquina del servidor.

INET_NTOA(expr)

Dada una dirección de red (4 o 8 bytes), retorna la representación de los cuatro valores representados en puntos como cadena:

```
mysql>SELECT INET_NTOA(3520061480);
->"209.207.224.40"
```

INET_ATON(expr)

Dada una representación de dirección de red con las 4 cifras separadas por puntos, retorna un entero que representa el valor numérico de la dirección. Las direcciones pueden ser de 4 o 8 bytes:

```
mysql>SELECT INET_ATON("209.207.224.40")
->3520061480
```

El número generado siempre se da según el orden de los bytes de la dirección IP; por ejemplo, el número anterior se ha calculado como 209*256^3+207*256^2+224*256+40.

MASTER_POS_WAIT(log_name,log_pos)

Bloquea hasta que el esclavo llega a la posición especificada en el log maestro durante la replicación. Si la información maestra no se inicializa, retorna NULL. Si el esclavo no está arrancando, bloqueará y esperará hasta que inicie y vaya hasta o supere la posición especificada. Si el esclavo ha pasado la posición especificada, retorna inmediatamente. El valor de retorno es el número de eventos del log que ha tenido que esperar para conseguir la posición especificada, o NULL en caso de error. Útil para el control de la sincronización MASTER-SLAVE, fue escrito originalmente para facilitar el testeo de la replicación.

FOUND_ROWS()

Retorna el número de filas que el último comando SELECT SQL_CALC_FOUND_ROWS... pudo retornar, si no se restringiera con LIMIT:

```
mysql>SELECT SQL_CALC_FOUND_ROWS * FROM tbl_name WHERE id>100 LIMIT 10;
mysql>SELECT FOUND_ROWS();
```

El segundo SELECT retornará un número indicando cúantas filas retornaría el primer SELECT si no hubiera sido escrito con la cláusula LIMIT.

Nota que si estás utilizando SELECT SQL_CALC_FOUND_ROWS ... MySQL ha de calcular todas las filas del resultado dado. Sin embargo, esto es más rápido que si no utilizaras LIMIT, ya que el resultado dado no necesita ser enviado al cliente.

SQL_CALC_FOUND_ROWS está disponible desde la versión 4.0.0 de MySQL.

6.3.7. Funciones para el uso en cláusulas GROUP BY

Si utilizas una función de agrupación en una sentencia que no contenga una cláusula GROUP BY, equivale a agrupar todas las filas.

COUNT(expr)

Retorna el total de valores no nulos en las filas recuperadas por la sentencia SELECT:

```
mysql>SELECT student.student_name,COUNT(*)
->FROM student,course
->WHERE student.student_id=course.student_id
->GROUP BY student_name;
```

COUNT(*) es algo diferente ya que retorna el número de filas restornadas, aunque incluyan un valor NULL.

COUNT(*) se optimiza para retornar muy rápido si SELECT recupera desde una tabla, no hay otras columnas retornadas, y no hay cláusula WHERE. Por ejemplo:

```
mysql>SELECT COUNT(*) FROM student;
```

COUNT(DISTINCT expr [,expr...])

Retorna el total de valores diferentes no nulos:

mysql>SELECT COUNT(DISTINCT results) FROM student;

En MySQL puedes obtener el número de diferentes combinaciones de expresiones que no contienen NULL dando una lista de expresiones. En ANSI SQL, podrías tener que hacer una concatenación de todas las expresiones dentro de COUNT(DISTINCT...).

AVG(expr)

Retorna la media de expr:

```
mysql>SELECT student_name, AVG(test_score)
->FROM student
->GROUP BY student_name;
```

MIN(expr)
MAX(expr)

Retorna el valor mínimo o máximo de expr. MIN() y MAX() pueden tomar un argumento de cadena: en tales casos retornan el mínimo o máximo valor de cadena. Puedes ver la sección 5.4.3

```
mysql>SELECT student_name, MIN(test_score), MAX(test_score)
->FROM student
->GROUP BY student_name;
```

SUM(expr)

Retorna la suma de expr. Nota que si el resultado de retorno no tiene filas, se retorna NULL!

STD(expr)
STDDEV(expr)

Retorna la desviación estándar de expr. Esta es una extensión del ANSI SQL. La forma STDDEV() de esta proporcionada por compatibilidad con Oracle.

BIT_OR(expr)

Retorna el OR bit a bit de todos los bits en expr. El cálculo se realiza con precisión de BIGINT (64 bits).

BIT_AND(expr)

Retorna el AND bit a bit de todos los bits en expr. El cálculo se realiza con precisión BIGINT (64 bits).

MySQL ha extendido el uso de GROUP BY. Puedes utilizar columnas o cálculos en las expresiones SELECT que no aparecen en la parte GROUP BY. Esto se mantiene para cualquier valor posible del grupo. Puedes utilizar esto para conseguir una mayor realización evitando la ordenación y agrupación de elementos innecesarios. Por ejemplo, no necesitas agrupar por customer.name en la siguiente consulta:

```
mysql>SELECT order.custid, customer.name, MAX(payments)
->FROM order,customer
->WHERE order.custid=customer.custid
->GROUP BY order.custid;
```

En ANSI SQL, deberías añadir customer.name en la cláusula GROUP BY. En MySQL, el nombre es redundante si no arrancas en modo ANSI.

No utilices esta funcionalidad si las columnas que omites en el GROUP BY no son únicas en el grupo! Puedes obtener resultados impredecibles.

En algunos casos, puedes utilizar MIN() y MAX() para obtener un valor específico de columna incluso si ésta es única. Lo siguiente da el valor de column desde la fila que contiene el valor más pequeño de la columna sort:

```
SUBSTR(MIN(CONCAT(RPAD(sort,6,' '),column)),7)
```

Puedes ver la sección 3.5.4 [example-Maximum-column-group-row]

Nota que si utilizas MySQL v3.22 (o anterior) o si estás tratando de seguir el ANSI SQL, no puedes utilizar las expresiones en las cláusulas GROUP BY o ORDER BY. Puedes trabajar sobre esta limitación utilizando un alias para la expresión:

```
mysql>SELECT id,FLOOR(value/100) AS val FROM tbl_name
->GROUP BY id,val ORDER BY val;
```

En la versión 3.23 de MySQL puedes hacer:

```
mysql>SELECT id, FLOOR(value/100) FROM tbl_name ORDER BY RAND();
```

6.4. Manipulación de datos: SELECT,INSERT,UPDATE,DELETE

6.4.1. Sintaxis SELECT

```
SELECT [STRAIGHT_JOIN]
    [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
    [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]  [HIGH_PRIORITY]
    [DISTINCT | DISTINCTROW | ALL]
    select_expression
    [INTO {OUTFILE|DUMPFILE} 'file_name'  export_options]
    [FROM table_references
    [WHERE where_definition]
    [GROUP BY {unsigned_integer | col_name | formula} [ASC|DESC] ]...
    [HAVING where_definition]
    [ORDER BY {unsigned_integer | col_name | formula} [ASC|DESC],... ]
    [LIMIT [offset,]rows ]
    [PROCEDURE procedure_name]
    [FOR UPDATE | LOCK IN  SHARE MODE]]
```

SELECT se utiliza para recuperar las filas seleccionados desde una o más tablas. La select_expression indica las columnas que quieres recuperar. SELECT puede utilizarse también para recuperar filas procesadas sin referencia a ninguna tabla. Por ejemplo:

```
mysql>SELECT 1+1;
->2
```

Todas las palabras claves utilizadas deben ser dadas en el orden exacto presentado más arriba. Por ejemplo, una cláusula HAVING debe hallarse después de cualquier cláusula GROUP BY y antes de cualquier cláusula ORDER BY.

- Una expresión SELECT puede venir dada a través de un alias utilizando AS. El alias se utiliza como el nombre de columna de la expresión y puede ser utilizado con las cláusulas ORDER BY y HAVING. Por ejemplo:

  ```
  mysql> SELECT CONCAT( last_name, ',  ',  first_name) AS full_name
          FROM mytable ORDER BY full_name;
  ```

- No se permite utilizar los alias de columnas en cláusulas WHERE, porque el valor de la columna puede no estar determinado cuando se ejecute la cláusula WHERE. Puedes ver A.5.4.
- La cláusula FROM table_references indica las tablas desde las que se recuperan las filas. Si nombras más de una tabla, estás realizando una join. Para más información sobre la sintaxis de JOIN, puedes ver la sección 6.4.1.1 [JOIN]. Para cada tabla especificada, puedes especificar opcionalmente un alias:

  ```
  table_name [[AS] alias] [USE INDEX(key_list)] [IGNORE INDEX (key_list)]
  ```

  Desde la versión 3.23.12 de MySQL, puedes dar recomendaciones sobre qué índice de debería utilizar MySQL al recuperar información desde una tabla. Esto es útil si EXPLAIN visualiza que MySQL está utilizando el índice equivocado. Especficando USE INDEX (key_list), puedes explicar a MySQL que utilice sólo uno de los índices especificados para encontrar filas en la tabla. La sintaxis alternativa IGNORE INDEX (key_list) puede ser utilizada para explicar a MySQL de no utilizar algún índice en particular. USE/IGNORE KEY son sinónimos para USE/IGNORE INDEX.

- Puedes referirte a una columna como col_name, tbl_name.col_name, o db_name.tbl_name.col_name. No precisas especificar prefijos de base de datos o tabla en una sentencia SELECT a no ser que sea una referencia ambigua. Puedes ver la sección 6.1.2. [Legal names].

- Una referencia a tabla puede ser nombrada con alias utilizando tbl_name [AS] alias_name:

  mysql>SELECT t1.name, t2.salary FROM employee AS t1, infor AS t2 WHERE t1.name=t2.name;

  mysql>SELECT t1.name, t2.salary FROM employee t1, info t2 WHERE t1.name=t2.name;

- Las columnas seleccionadas para la salida pueden ser referenciadas en cláusulas GROUP BY y ORDER BY utilizando nombres de columna, alias de columnas, o posiciones de columna. Las posicionesd de columna empiezan en 1:

  mysql> SELECT college,region, seed FROM tournament ORDER BY region, seed;
  mysql> SELECT college,region AS r, seed AS s FROM tournament ORDER BY r, s;

  Para ordenar en orden inverso, añade la palabra clave DESC al nombre de la columna de la cláusula ORDER BY por la que estás ordenando. El valor por defecto es orden ascendente, que puede ser especificado con la palabra clave ASC.

- Puedes usar en la cláusula WHERE cualquiera de las funciones que soporta MySQL. Puedes ver la sección 6.3 [Function].

- La cláusula HAVING puede referirse a cualquier columna o alias nombrado en select_expression. Se aplica en último término, justo antes que los datos se envíen a cliente, sin optimización. No utilices HAVING para elementos que deberían estar en la cláusula WHERE. Por ejemplo, no escribas esto:

mysql> SELECT col_name FROM tbl_name HAVING col_name>0;

Cámbialo por:

mysql> SELECT col_name FROM tbl_name WHERE col_name >0;

En la versión 3.22.5 de MySQL y posteriores, también puedes escribir consultas como esta:

SELECT user,MAX(salary) FROM users GROUP BY user HAVING MAX(salary)>10;

En versiones anteriores de MySQL, puedes cambiar esto por:

SELECT user,MAX(salary) AS sum FROM users GROUP BY user HAVING sum>10;

- Las opciones DISTINCT,DISTINCTROW y ALL especifican cuando las filas duplicadas deberían ser retornadas. El valor por defecto es ALL, de modo que todas las filas coincidentes se retornan. DISTINCT y DISTINCTROW son sinónimos y especifican que las filas duplicadas en el resultado deberían ser obviadas.
- Todas las opciones empezando por SQL_, STRAIGHT_JOIN, y HIGH_PRIORITY son extensiones de MySQL sobre ANSI SQL.

- HIGH_PRIORITY dará al SELECT mayor prioridad que una sentencia que actualice una tabla. Debes utilizarlo sólo en consultas que son muy rápidas y deben ser realizadas una vez. Una consulta SELECT HIGH_PRIORITY arrancará si la tabla se bloquea para lectuar incluso si hay una sentencia de actualización que espera para que la tabla se libere.
- SQL_BIG_RESULT puede ser utilizada con GROUP BY o DISTINCT para indicar al optimizador que el resultado dado tendrá varias filas. En este caso, MySQL utilizará directamente tablas temporales guardadas en disco si es necesario. MySQL también preferirá, en este caso, ordenar para hacer una tabla temporal con una clave en los elementos GROUP BY.
- SQL_BUFFER_RESULT forzará el resultado a ser introducido en una tabla temporal. Esto ayudará a MySQL a liberar los bloqueos de tabla pronto y ayudará en casos en los que tome un largo rato enviar el resultado al cliente.
- SQL_SMALL_RESULT, una opción específica de MySQL, puede ser utilizada con GROUP BY o DISTINCT para indicar al optimizador que el resultado será pequeño. En este caso, MySQL utilizará tablas temporales rápidas para almacenar la tabla resultante en vez de usar la ordenación. En la versión 3.23 de MySQL no debería ser necesario, normalmente.
- SQL_CALC_FOUND_ROWS indica a MySQL que calcule el número de filas que habría en el resultado, ignorando cualquier cláusula LIMIT. El número de filas puede ser obtenido con SELECT FOUND_ROWS(). Puedes ver la sección 6.3.6.2 [miscellaneous functions].
- SQL_CACHE indica MySQL que almacene el resultado de la consulta en el cache de consulta. Puedes ver la sección 6.9 [Query cache].
- SQL_NO_CACHE indica a MySQL que no permita el almacenamiento del resultado de la consulta en el caché de consultas. Puedes ver la sección 6.9 [Query cache].
- Si utilizas GROUP BY, las filas resultantes se almacenarán de acuerdo con GROUP BY como si tuvieras un ORDER BY sobre todos los campos del GROUP BY. MySQL ha extendido GROUP BY para que puedas especificar ASC o DESC para GROUP BY:

  SELECT a, COUNT(b) FROM test_table GROUP BY a DESC;

- MySQL ha extendido el uso de GROUP BY para permitirte seleccionar campos que no se mencionan en la cláusula GROUP BY. Si no consigues los resultados esperados con tu consulta, lee la descripción de GROUP BY. Puedes ver la sección 6.3.7 [group by functions].
- STRAIGHT_JOIN fuerza el optimizador a unir las tablas en el orden en el que se listan en la cláusula FROM. Puedes utilizarla para incrementar la velocidad una consulta si el optimizador las une en un orden no-óptimo. Puedes ver la sección 5.2.1 [EXPLAIN].
- La cláusula LIMIT puede ser utilizada para restringir el número de filas retornados por la sentencia SELECT. LIMIT toma uno o dos argumentos numéricos. Los argumentos deben ser constantes enteras. Si se dan dos argumentos, el primero especifica el número de orden (off-set) de la primera fila a retornar, el segundo especifica el número máximo de filas a retornar. El número de la fila inicial es 0 (no 1):

  mysql>SELECT * FROM table LIMIT 5,10; # recupera filas 6 a 15

  Para recuperar todas las filas desde un determinado número hasta el final, puedes utilizar –1 para el segundo parámetro:

  mysql>SELECT * FROM table LIMIT 95,-1; # recupera filas 96-última

  Si se da un argumento, indica el máximo número de filas a retornar.

  mysql>SELECT * FROM table LIMIT 5; # recupera las primeras 5 filas

En otras palabras, LIMIT n=LIMIT 0,n.

- La forma SELECT...INTO OUTFILE 'file-name' de SELECT escribe las filas seleccionadas en un archivo. El archivo se crea en el host del servidor y no puede existir anteriormente (entre otras cosas, esto previene a las tablas de la base de datos y archivos tales como 'etc/passwd' de ser destruídas). Debes disponer del privilegio FILE en el host del servidor para utilizar esta forma de SELECT.

  La forma SELECT...INTO OUTFILE es utilizado principalmente para permitirte volcar rápidamente una base de datos en una máquina de servidor. Si quieres crear el archivo en cualquier otro host que el del servidor, no puedes utilizar SELECT...INTO OUTFILE. En este caso deberías utilizar un programa cliente como mysqldump –tab o mysql –e "SELECT ...." >outfile para generar el archivo.

  SELECT ... INTO OUTFILE es el complemento a LOAD DATA INFILE; el sintaxis par la parte export_options de la sentencia consiste en las mismas cláusulas FIELDS y LINES que se utilizan con la sentencia LOAD DATA INFILE. Puedes ver la sección 6.4.9 [LOAD DATA].

  En el archivo de texto resultante, sólo los siguientes caracteres son escapados por el carácter SCAPED BY:

  - El carácter SCAPED BY.
  - El primer carácter en FIELDS TERMINATED BY.
  - El primer carácter en LINES TERMINATED BY.

  Adicionalmente, el carácter ASCII 0 se convierte a SCAPED BY seguido por 0 (ASCII 48).

  La razón para lo anterior es que debes escapar cualquier carácter de FIELDS TERMINATED BY, SCAPED BY, o LINES TERMINATED BY para ser capaz fiablemente de leer el archivo. ASCII es escapado para verlo mejor desde algunos paginadores.

  Como el archivo resultante no tiene que concordar con la sintaxis SQL, no es necesario que nada más sea escapado. Aquí tenemos un ejemplo para coger un archivo en el formato utilizado por muchos programas antiguos:

  ```
  SELECT a,b,a+b INTO OUTFILE "/tmp/result.text"ç
  FIELDS TERMINATED BY ',  ' OPTIONALLY ENCLOSED BY '"'
  LINES TERMINATED BY "\n"
  FROM test_table;
  ```

- Si utilizas INTO DUMPFILE en vez de INTO OUTFILE, MySQL sólo escribirá una fila en el archivo, sin terminaciones de columna ni línea y sin escapar. Esto es útil si quieres almacenar un blob en un archivo.
- Nota que cualquier archivo creado por INTO OUTFILE y INTO DUMPFILE será leíble por todos los usuarios! La razón es que el servidor MySQL no puede crear un archivo propiedad de un usuario que el usuario que está corriendo (no deberías utilizar nunca mysqld como root), el archivo tiene que ser leíble en palabras [word readable] de modo que puedas recuperar las columnas.
- Si estás utilizando FOR UPDATE en un manejador de tablas con bloqueos de página/fila, las filas examinadas se bloquearán contra escritura.

6.4.1.1. Sintaxis JOIN

MySQL soporta las siguientes sintaxis JOIN para el uso en sentencias SELECT:

table_reference, table_reference
table_reference [CROSS] JOIN table_reference
table_reference INNER JOIN table_reference join_condition
table_reference STRAIGHT_JOIN table_reference
table_reference LEFT [OUTER] JOIN table_reference join_condition
table_reference LEFT [OUTER] JOIN table_reference
table_reference NATURAL [LEFT[OUTER] JOIN table_reference
{oj table_reference LEFT OUTER JOIN table_reference ON conditional_expr}
table_reference RIGHT [OUTER] JOIN table_reference join_condition
table_reference RIGHT [OUTER] JOIN table_reference
table_reference NATURAL [RIGHT[OUTER]] JOIN table_reference

Donde table_reference se define como:

table_name [[AS] alias] [USE INDEX (key_list)] [IGNORE INDEX (key_list)]

y join_condition se define como:

ON conditional_expr |
USING (column_list)

Nunca deberías tener condiciones en la parte ON que se utilicen para restringir las filas que deberían haver en el resultado. Si quieres restringir las filas del resultado, tienes que hacerlo con la cláusula WHERE.

Nota que las versiones anteriores a la 3.23.17, el INNER JOIN no tomaba una join_condition! La última sintaxis LEFT OUTER JOIN vista existe sólo por compatibilidad con ODBC:

- Una referencia a tabla puede ser llamada con alias utilizando tbl_name AS alias_name o tbl_name alias_name.

mysql> SELECT t1.name,t2.salary FROM employee AS t1, info AS t2 WHERE t1.name=t2.name;

- El condicional ON es cualquier condicional de forma que podría ser usado en una cláusula WHERE.
- Si no hay registros coincidentes en la tabla derecha en la parte ON o USING en un LEFT JOIN, una fila con todas las columnas con valores NULL se utiliza para la tabla derecha. Puedes utilizar este hecho para encontrar registros en una tabla que no tienen correspondencias en otra tabla.

```
mysql> SELECT table1.* FROM table1
       LEFT JOIN table2 ON table1.id=table2.id
       WHERE table2.id IS NULL;
```

Este ejemplo busca todas las filas en table1 con un id de valor que no está presente en table2 (esto es, todas las filas en table1 sin correspondencia en table2). Esto asume que table2.id se declara como NOT NULL, desde luego. Puedes ver la sección 5.2.6. [LEFT JOIN optimisation].

- La cláusula USING (column_list) nombra una lista de columnas que deben existir en ambas tablas. Una cláusula USING tal como:
  A LEFT JOIN B USING (C1,C2,C3,...)

  se define como semáticamente idéntica a una expresión ON como esta:

  A.C1=B.C1 AND A.C2=B.C2 AND A.C3=B.C3,...

- La NATURAL [LEFT] JOIN entre dos tablas se define como semáticamente equivalente a un INNER JOIN o LEFT JOIN utilizando la cláusula USING que indique todos los nombres de columnas que existen en ambas tablas.
- INNER JOIN y , (coma) son semáticamente equivalentes. Ambos dan una conexión completa entre las tablas utilizadas. Normalmente, especificas cómo deberían estar enlazadas las tablas en la condición WHERE.
- RIGHT JOIN trabaja análogamente a LEFT JOIN. Para mantener el código portable entre bases de datos, se recomienda utilizar LEFT JOIN en vez de RIGHT JOIN.
- STRAIGHT_JOIN es idéntico a JOIN, excepto que la tabla izquierda se lee siempre antes que la tabla derecha. Esto puede ser utilizado para aquellos
  (pocos) casos donde el optimizador de joins pone las tablas en orden incorrecto.
- Como en MySQL v.3.23.12, puedes dar sugerencias sobre qué índice debería utilizar MySQL al recuperar información desde una tabla. Esto es útil si EXPLAIN visualiza que MySQL está utilizando el índice incorrecto. Especificando USE INDEX (key_list), puedes explicar MySQL que utilice sólo uno de los índices especificados en la tabla. La sintaxis alternativa IGNORE INDEX (key_list) puede ser utilizada para explicar a MySQL que no utilice algunos índices particulares. USE/IGNORE INDEX son sinónimos de USE/IGNORE INDEX.

Algunos ejemplos:

```
mysql>SELECT * FROM table1,table2, WHERE table1.id=table2.id;
mysql>SELECT * FROM table1 LEFT JOIN table2 ON table1.id=table2.id;
mysql>SELECT * FROM table1 LEFT JOIN table2 USING (id);
mysql>SELECT * FROM table1 LEFT JOIN table2 ON table1.id=table2.id
                        LEFT JOIN table3 ON tabl2.id=table3.id;
mysql>SELECT * FROM table1 USE INDEX (key1,key2)
                        WHERE key1=1 AND key2=2 AND key3=3;
mysql>SELECT * FROM table1 USE INDEX (key3)
                        WHERE key1=1 AND key2=2 AND key3=3;
```

Puedes ver la sección 5.2.6. [optimización LEFT JOIN].

6.4.1.2. Sintaxis de UNION

```
SELECT ....
UNION [ALL]
SELECT ...
   [UNION
        SELECT ...]
```

UNION se ha implementado en la versión 4.0.0.

UNION se utiliza para combinar el resultado de varias sentencias SELECT en un solo resultado. Las columnas listadas en la en la porción select_expression del SELECT debería tener el

mismo tipo. Los nombres de columna en la primera consulta SELECT se utilizará como nombres de columna en los resultados retornados.

Los comandos SELECT son comandos de selección normales, pero con las siguientes restricciones:

- Sólo el último SELECT puede tener INTO OUTFILE.

Si no utilizas la palabra clave ALL para la UNION, todas las filas retornadas serán únicas, como si hubieras hecho un DISTINCT para el resultado final. Si especificas ALL, obtendrás todas las filas coincidentes desde todas las sentencias SELECT utilizadas.

Si quieres utilizar un ORDER BY para el resultado total de la unión, deberías utilizar paréntesis:

(SELECT a FROM table_name WHERE a=10 AND B=1 ORDER BY a LIMIT 10) UNION (SELECT a FROM table_name WHERE a=11 AND B=2 ORDER BY a LIMIT 10) ORDER BY a;

6.4.2. Sintaxis de HANDLER

HANDLER table_name OPEN [AS alias]

HANDLER table_name READ index_name {= | >= | <= | <} (value1,value2,...)
[WHERE ...] [LIMIT ...]

HANDLER tbl_name READ index_name {FIRST | NEXT | PREV | LAST}
[WHERE ...] [LIMIT ...]

HANDLER table_name CLOSE

La sentencia HANDLER provee acceso directo a la interficie del handler de la tabla MyISAM.

La primera forma de la sentencia HANDLER abre una tabla, haciéndola accesible vía las subsiguientes sentencias HANDLER...READ. Este objeto de tabla no se comparte con otras llamadas y no se cerrará hasta que la llamada indique HANDLER table_name CLOSE o la llamada muera.

La segunda forma lee una fila (o más, especificado por la cláusula LIMIT) donde el índice especificado se ajusta a la condición y la condición WHERE se cumple. Si el índice consiste en varias partes (separa sobre varias columnas) los valores se especifican en una de elementos separados por coma, aportando sólo valores para los que las pocas primeras columnas es posible.

La tercera forma lee una fila (o más, especificado por la cláusula LIMIT) desde la tabla en orden marcado por el índice, ajustándose a la condición WHERE.

La cuarta forma (sin especificación de índice) lee una fila (o más, especficado en la cláusula LIMIT) desde la tabla en orden natural de los registros (como se insertaron en el archivo) ajustándose a la condición WHERE. Es más rápido que HANDLER table_name READ index_name cuando se desea el escaneo la tabla completa.

HANDLER...CLOSE cierra la tabla que se abrió con HANDLER...OPEN.

HANDLER es de algún modo una sentencia de bajo nivel. Por ejemplo, no permite consistencia. Esto es, HANDLER...OPEN no echa un vistazo de la tabla, ni la bloquea. Esto significa que después que un HANDLER...OPEN es realizado, los datos de la tabla pueden ser modificados (por esta o cualquier otra llamada) y estas modificaciones pueden aparecer sólo parcialmente en escaneos con HANDLER...NEXT o HANDLER...PREV.

Las razones para utilizar esta interficie en vez de la normal de SQL son:

- Es más rápido que el SELECT porque:
  - Un manejador de tabla determinado se aloja a la apertura del HANDLER.
  - Menos parseo implicado.
  - No hay desbordamiento del optimizador y chequeo de la consulta.
  - La tabla usada no debe ser bloqueada entre dos peticiones de handler.
  - La interficie del handler no tiene que proveer una vista consistente de los datos (por ejemplo, se permiten dirty-reads), que permiten al manejador de la tabla realizar una optimización que SQL no lo permite normalmente.
- Esto hace mucho más sencillo para portar aplicaciones utilizan interficies del estilo ISAM para MySQL.
- Esto le permite a uno transversar [traverse] una base de datos de una manera que no es sencillo (en algunos casos imposible) hacer con SQL. La interficie del manejador es la vía más natural para mirar los datos cuando se trabajan con aplicaciones que proveen una interficie de usuario de base de datos interactiva.

6.4.3. Sintaxis INSERT

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
    [INTO] table_name [(col_name,...)]
    VALUES ((expression | DEFAULT),...),(...),...

INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
    [INTO] table_name [(col_name,...)]
    SELECT ...

INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
    [INTO] table_name [(col_name,...)]
    SET col_name=(expression | DEFAULT),...
```

INSERT inserta nuevos registros en una tabla existente. La forma INSERT...VALUES de la sentencia inserta filas basada en valores explícitamente especificados. La forma INSERT...SELECT inserta las filas seleccionadas desde otra tabla o tablas. La forma INSERT...VALUES con múltiples listas de valores se soporta en MySQL 3.22.5 o posterior. La sintaxis col_name=expression se soporta en MySQL 3.22.10 o posterior.

- Si no especificas ninguna lista de columnas para INSERT...VALUES o INSERT...SELECT, los valores para todas las columnas han de ser dados en la lista VALUES() o por SELECT. Si no sabes el orden de las columnas en la tabla, utiliza DESCRIBE table_name para encontrarlo.
- Cualquier columna no dada explícitamente se sitúa en el valor por defecto. Por ejemplo, si especificas una lista de columnas que no nombra todas las columnas en la tabla, las columnas no nombradas se situarán en sus valores por defecto. La asignación de valores por defecto se describe en la sección 6.5.3. [CREATE TABLE]

  También puedes utilizar la palabra clave DEFAULT para dar el valor por defecto a las

columnas. (Nuevo en MySQL 4.0.3.) Esto hace mucho más sencillo escribir sentencias INSERT que asignen valores a todas las columnas excepto unas pocas, porque ello te permite evitar de escribir una lista VALUES incompleta (una lista que no incluye un valor para cada columna en la tabla). De cualquier otro modo, deberías escribir la lista de nombres de columna correspondiente para cada valor de la lista VALUES().

MySQL siempre tiene un valor por defecto para todos los campos. Esto es algo que se impone en MySQL para poder trabajar con tablas transaccionales y no transaccionales.

Nuestra visión es que el chequeo de contenido de campos debería ser realizado en la aplicación y no en el servidor de la base de datos.

- Una expresión puede referirse a cualquier columna que fue asignada antes en una lista de valores. Por ejemplo, puedes decir esto:

  mysql> INSERT INTO table_name (col1,col2) VALUES (15,col1*2);

  Pero no esto:

  mysql> INSERT INTO table_name (col1,col2) VALUES (col2*2,15);

- Si especificas la palabra clave LOW_PRIORITY, la ejecución del INSERT se retrasa hasta que no haya otros clientes leyendo desde la tabla. En este caso, el cliente ha de esperar hasta que la inserción se completa, lo que puede tomar un largo rato si la tabla tiene un fuerte uso. Esto contrasta con INSERT DELAYED, que permite al cliente continuar al mismo tiempo. Puedes ver la sección 6.4.4. [INSERT DELAYED]. Nota que LOW_PRIORITY no debería ser utilizado normalmente con tablas MyISAM ya que desactiva las inserciones concurrentes. Puedes ver la sección 7.1 [MyISAM].
- Si especificas la palabra clave IGNORE y un INSERT con varios valores de filas, cualquier fila que duplique una clave PRIMARY o UNIQUE en la tabla se ignora y no se inserta. Si no especificas IGNORE, la inserción aborta si hay cualquier fila que duplica un valor clave existente. Puedes determinar con la función de la API de C mysql_info() cuántas filas se insertaron en la tabla.
- Si MySQL se configuró utilizando la opción DONT_USE_DEFAULT_FIELDS, la sentencia INSERT generan un error a menos que especifiques explícitamente valores para todas las columnas que requieran un valor no nulo. Puedes ver la sección 2.3.3. [configure options].
- Puedes encontrar el valor usado en una columna AUTO_INCREMENT con la función mysql_insert_id. Puedes ver la sección 8.4.3.30 [mysql_insert_id()].

Si utilizas una sentencia INSERT...SELECT o INSERT...VALUES con múltiples listas de valores, puedes utilizar la función de la API de C mysql_info() para obtener información sobre la consulta. El formato del texto de la información es visto aquí:

Records: 100 Duplicates: 0 Warnings: 0

Duplicates indica el número de filas que no pudieron ser insertados porque duplicarían algún valor clave. Warnings indica el número de intentos de insertar valores de columna que tuvieron problemas en algún sentido. Las warnings pueden ocurrir bajo cualquiera de las siguientes condiciones:

- Insertar un NULL en una columna que se ha declarado NOT NULL. La columna se sitúa en su valor por defecto.
- Situando una columna numérica a un valor que supera el rango permitido. El valor se ajusta al máximo del rango.

- Situando una columna numérica a un valor tal como '10.34 a'. La basura sobrante se quita y permanece la parte numérica. Si el valor no tiene sentido como número, la columna valdrá 0.
- Insertando una cadena dentro de una columna CHAR, VARCHAR, TEXT o BLOB que exceda su máxima longitud. El valor se trunca a la longitud máxima de la cadena.
- Insertar un valor en una columna de fecha o tiempo que es ilegal para el tipo de columna. La columna se sitúa a valor cero adecuado para el tipo.

6.4.3.1. Sintaxis INSERT...SELECT

INSERT [LOW_PRIORITY] [IGNORE] [INTO] table_name [(column_list)] SELECT ...

Con la sentencia INSERT...SELECT puedes instertar rápidamente varias filas en una tabla desde varias tablas.

INSERT INTO tblTemp2 (fldID) SELECT tblTemp1.flOrder_ID FROM tblTemp1 WHERE tblTemp1.flOrder_ID >100;

Las siguientes condiciones se mantienen para una sentencia INSERT...SELECT:

- La tabla de destino de la sentencia INSERT no puede aparecer en la cláusula FROM de la parte SELECT de la consulta porque se prohibe en ANSI SQL seleccionar desde la misma tabla en la que se está insertando. (El problema es que el SELECT posiblemente podría encontrar registros que fueron insertados previamente durante la misma ejecución. Al utilizar las cláusulas de subselección, la situación podría fácilmente ser muy confusa).
- Las columnas AUTO_INCREMENT trabajan como es usual.
- Puedes utilizar la función de la API de C mysql_info() para obtener la información sobre la consulta. Puedes ver la sección 6.4.3. [INSERT].
- Para asegurarse que el update log/binary log puede ser utilizado para re-crear las tablas originales, MySQL no permitirá inserciones concurrentes durante el INSERT...SELECT.

Desde luego puedes utilizar REPLACE en vez de INSERT para sobreescribir registros antiguos.

6.4.4. Sintaxis INSERT DELAYED

INSERT DELAYED ...

LA opción DELAYED para la sentencia INSERT es una opción específica de MySQL que es muy útil si tienes clientes que no pueden esperar para que el INSERT se complete. Este es un problema común cuando utilizas MySQL para loginarte y también utilizas periódicamente SELECT y UPDATE que se toman su tiempo para completarse. DELAYED fue introducido en la versión 3.22.15 de MySQL de extensión al ANSI SQL92.

INSERT DELAYED sólo trabaja con tablas ISAM y MyISAM. Nota que como las tablas MyISAM soportan SELECT y INSERT concurrentes, si no hay bloques libres en el medio del archivo, muy raramente necesitarás utilizar INSERT DELAYED con MyISAM. Puedes ver la sección 7.1 [MyISAM].

Cuando utilizas INSERT DELAYED, el cliente recibirá un OK una vez y la fila se insertará cuando la tabla no esté en uso por cualquier otra llamada.

Otro mayor beneficio de utilizar INSERT DELAYED es que inserta desde varios clientes empaquetados y escritos en un mismo bloque. Esto es mucho más rápido que hacer varias inserciones por separado.

Nota que actualmente las filas en cola sólo se almacenan en la memoria hasta que se insertan en la tabla. Esto significa que si matas mysqld de forma drástica (kill -9) o si mysqld muere inesperadamente, cualquier fila en cola que no se escribiera en el disco se perderá!

Lo siguiente describe en detalle qué pasa cuando utilizas DELAYED para insertar o reemplazar. En esta descripcion, la "llamada" es el ataque que recibe un comando INSERT DELAYED, y "handler" es el ataque que gestiona todas las sentencias INSERT DELAYED para una tabla particular.

- Cuando una llamada local ejecuta un DELAYED para una tabla, un handler es creado para procesar todas las sentencias DELAYED para la tabla, si no existe ningún handler.
- La llamada chequea si el handler ya ha recibido un bloqueo DELAYED; sino, explica a la llamada del handler que lo haga. El bloqueo DELAYED puede ser obtenido aún si otras llamadas tienen un bloqueo READ o WRITE en la tabla. Sin embargo, el handler esperará a todos los bloqueos ALTER TABLE o FLUSH TABLES para asegurarse que la estructura de la tabla está actualizada.
- La llamada ejecuta la sentencia INSERT, pero en vez de escribir la fila en la tabla, pone una copia de la fila final en una cola que se gestiona por la llamada del handler. Cualquier error de sintaxis se notifica por la llamada y es reportada por el programa cliente.
- El cliente no puede reportar el número de duplicados del valor AUTO_INCREMENT para la fila resultante; no puede obtenerlos del servidor, porque el INSERT retorna antes de que la operación de inserción se complete. Si utilizas la API de C, la función mysql_info() no retorna nada significativo, por la misma razón.
- El log de actualización se actualiza por la llamada del handler cuando la fila se inserta en la tabla. En caso de inserciones de múltiples filas, la actualización del log se actualiza cuando se inserta la primera fila.
- Después de escribir delayed_insert_limit filas, el handler chequea si existen SELECT's pendientes. Si es el caso, le permite ejecutarse antes de continuar.
- Cuando el handler no tiene más filas en su cola, la tabla se desbloquea. Si no hay nuevos comandos INSERT DELAYED en petición espera hasta que exista algo en la cola. Esto se hace para asegurarse que el servidor mysqld no utilice toda la memoria para la cola de inserción retardada.
- La llamada del handler visualizará en la lista de procesos MySQL con delayed_insert en la columna command. Se matará si ejecutas un comando FLUSH TABLES o lo matas con KILL thread_id. Sin embargo, primero almacenará todas las filas en cola en la tabla antes de salir. Durante este tiempo no aceptará nuevos comandos INSERT desde otra llamada. Si ejecutas un comando INSERT DELAYED después de éste, se creará una nueva llamada al handler.
  Nota que lo anterior significa que los comandos INSERT DELAYED tienen mayor prioridad que los comandos normales INSERT si hay un comando INSERT DELAYED ejecutándose! Otros comandos de actualización deberán esperar hasta que la cola INSERT DELAYED se vacíe, alguien mate la llamada del handler (con KILL thread_id), o alguien ejecute FLUSH TABLES.
- Las siguientes variables de estado aportan información sobre los comandos INSERT DELAYED:
  - Delayed_insert_threads: Número de llamadas al handler.
  - Delayed_writes: Número de filas escritas con INSERT DELAYED.
  - Not_flushed_delayed_: Número de filas esperando ser escritas.

Puedes ver estas variables utilizando la sentencia SHOW STATUS o ejecutando un comando mysqladmin extended-status.

Nota que INSERT DELAYED es más lento que el INSERT normal si la tabla no está en uso. Además existe un desbordamiento adicional para el servidor para manejar una llamada separada para cada tabla en la que uses INSERT DELAYED. Esto significa que sólo deberías usar INSERT DELAYED cuando estás seguro que lo necesitas!

6.4.5. Sintaxis UPDATE

```
UPDATE [LOW_PRIORITY] [IGNORE] table_name
    SET col_name1=expr1 [,col_name2=expr2, ....]
    [WHERE where_definition]
    [LIMIT #]
```

UPDATE actualiza las columnas en filas existentes de una tabla con nuevos valores. La cláusula SET indica qué columnas modificar y qué valores deberían darse. La cláusula WHERE, si se da, especifica qué filas deberían ser actualizadas. En cualquier otro caso, todas las filas se actualizan. Si la cláusula ORDER BY se especifica, los registros se actualizarán en el orden especificado.

Si especificas la palabra clave LOW_PRIORITY, la ejecución del UPDATE se retrasa hasta que no existan otros clientes leyendo la tabla.

Si especificas la palabra clave IGNORE, la sentencia de actualización no abortará aunque se den errores de claves duplicadas durante la actualización. Las filas que causasen conflictos no se actualizarían.

Si accedes a una columna de la tabla table_name en una expresión, UPDATE utiliza en valor actual de la columna. Por ejemplo, la siguiente sentencia sitúa age a uno más que su valor actual:

```
mysql> UPDATE persondata SET age=age+1;
```

Las asignaciones de UPDATE se evalúan de izquierda a derecha. Por ejemplo, la siguiente sentencia dobla la columna age, y luego la incrementa:

```
mysql> UPDATE persondata SET age=age*2, age=age+1;
```

Si sitúas un valor de columna al actual, MySQL indica que esto no lo actualiza.

UPDATE retorna el número de filas que se cambiaron. En la versión MySQL 3.22 o posterior, la función de la API de C mysql_info() retorna el número de filas coincidentes y actualizadas y el número de alertas que ocurrieron durante la actualización.

En MySQL versión 3.23, puedes utilizar LIMIT # para asegurarte que sólo un número dado de registros son actualizados.

6.4.6. Sintaxis DELETE

```
DELETE [LOW_PRIORITY |QUICK] FROM table_name
    [WHERE where_definition]
```

```
    [ORDER BY...]
    [LIMIT rows]
```

o

```
DELETE [LOW_PRIORITY |QUICK] table_name[.*] [, table_name[.*],...]
    FROM table-reference
    [WHERE where_definition]
```

o

```
DELETE [LOW_PRIORITY |QUICK]
    FROM table_name[.*] [, table_name[.*],...]
    [USING table_reference]
    [WHERE where_definition]
```

DELETE borra las filas de la tabla table_name que satisfacen la condición dada por where_definition y retorna el número de registros borrados.

Si ejecutas un DELETE sin cláusula WHERE, se borran todos los registros. Si esto lo ejecutas en modo AUTOCOMMIT, funciona como TRUNCATE. Puedes ver la sección 6.4.7. [TRUNCATE]. En MySQL 3.23 DELETE sin WHERE retornará cero como número de registros afectados.

Si realmente quieres saber cuántos registros se borran cuando estás borrando todas las filas, y esperas sufrir una penalización de velocidad, puedes utilizar una sentencia DELETE de este modo:

```
mysql> DELETE FROM table_name WHERE 1>0;
```

Nota que esto es mucho más lento que DELETE FROM table_name sin cláusula WHERE, porque borra un registro cada vez.

Si especificas la palabra clave LOW_PRIORITY, la ejecución del DELETE se retrasará hasta que no haya clientes leyendo desde la tabla.

Si especificas la palabra QUICK, entonces el manejador de la tabla no fusionará índices durante el borrado, lo que puede acelerar ciertos tipos de borrados.

En tablas MyISAM, los registros borrados se mantienen en una lista encadenada y las subsiguientes operaciones INSERT reusan las viejas posiciones de los registros. Para reclamar espacio en desuso y reducir los tamaños de archivo, utiliza OPTIMIZE TABLE o la utilidad myisamchk para reorganizar tablas. OPTIMIZE TABLE es mas sencillo, pero myisamchk es más rápido. Puedes ver la sección 4.5.1. [OPTIMIZE TABLE] y la sección 4.4.6.10 [Optimisation].

El primer borrado multi-tabla se soporta desde MySQL 4.0.0. El segundo formato de borrado multi-tabla se soporta desde MySQL 4.0.2.

La idea es que sólo los registros coincidentes de las tablas listadas antes del FROM o de USING se borran. El efecto es que puedes borrar filas desde varias filas de una vez y además tienes tablas adicionales que se usan para búsqueda.

El .* después del nombre de la tabla se utiliza por compatibilidad con Access:

DELETE t1,t2 FROM t1,t2,t3 WHERE t1.id=t2.id AND t2.id=t3.id

o

DELETE t1,t2 USING t1,t2,t3 WHERE t1.id=t2.id AND t2.id=t3.id

En los anteriores casos borramos las filas coincidentes sólo de las tablas t1 y t2.

ORDER BY utilizando múltiples tablas en la sentencia DELETE se soporta en MySQL 4.0.

Si se utiliza una cláusula ORDER BY, las filas se borrarán en tal orden. Esto sólo es realmente útil en combinación con LIMIT. Por ejemplo:

```
DELETE FROM somelog
    WHERE user="jcole"
    ORDER BY timestamp
    LIMIT 1
```

Esto borrará la entrada más antigua (ordenando por timestamp) donde el registro coincide con la cláusula WHERE.

La opción específica de MySQL LIMIT rows del DELETE explica al servidor el máximo número de filas a ser borradas antes que el control se retorne al cliente. Esto puede utilizarse para asegurarse que un comando específico de DELETE no se toma demasiado tiempo. Simplemente puedes repetir el comando DELETE hasta que el número de filas afectadas sea menor que el valor LIMIT.

6.4.7. Sintaxis TRUNCATE

TRUNCATE TABLE table_name

En 3.23 TRUNCATE TABLE es mapeada a COMMIT; DELETE FROM table_name. Puedes ver la sección 6.4.6 [DELETE].

TRUNCATE TABLE difiere de DELETE FROM... en los siguientes sentidos:

- Las operaciones de truncado vacían y re-crean la tabla, lo que es mucho más rápido que borrar las filas una por una.
- No está a salvo de transacciones; obtendrás un error si tienes una transacción en activo o un bloqueo activo de la tabla.
- No retorna el número de registros borrados.
- En la medida en que el archivo de la definición de tabla 'table_name.frm' es válido, la tabla puede ser re-creada, incluso si los datis de los archivos de índice están corrompidos.

TRUNCATE es una extensión SQL de Oracle.

6.4.8. Sintaxis REPLACE

```
REPLACE [LOW_PRIORITY |DELAYED]
    [INTO] table_name [(col_name,...)]
    VALUES (expression,...), (...),...
```

o

```
REPLACE [LOW_PRIORITY |DELAYED]
    [INTO] table_name [(col_name,...)]
    SELECT ...
```

o

```
REPLACE [LOW_PRIORITY |DELAYED]
    [INTO] table_name [(col_name,...)]
    SET col_name=expression, col_name=expression,...
```

REPLACE trabaja exactamente como INSERT, excepto que si un registro antiguo de la tabla tiene el mismo valor que el registro nuevo en un índice UNIQUE o PRIMARY KEY, el registro antigui se borra antes de insertar el nuevo. Puedes ver la sección 6.4.3. [INSERT].

En otras palabras, no puedes acceder a los valores del registro antiguo desde la sentencia REPLACE. En algunas versiones antiguas de MySQL parecía que podías hacerlo, pero se trataba de un bug que ha sido corregido.

Cuando utilizas el comando REPLACE, mysql_affected_rows() retornará 2 si la nueva fila remplaza un registro antiguo. Esto se debe a que un registro se insertó y el duplicado se borró.

Este hecho simplifica el hecho de determinar los casos en los que REPLACE añadió o reemplazó un registro: chequea cuándo el valor affected_rows vale 1 (añadido) o 2 (reemplazado).

Nota que a menos que utilices un índice UNIQUE o PRIMARY KEY, utilizando un comando REPLACE no tiene sentido, ya que simplemente realizaría un INSERT.

6.4.9. Sintaxis LOAD DATA INFILE

```
LOAD DATA [LOW_PRIORITY|CONCURRENT] [LOCAL] INFILE 'filename.txt'
    [REPLACE|IGNORE]
    INTO TABLE tbl_name
    [FIELDS
        [TERMINATED BY '\t']
        [OPTIONALLY ENCLOSED BY '']
        [ESCAPED BY '\\']
    ]
    [LINES TERMINATED BY '\n']
    [IGNORE number LINES]
    [(col_name,...)]
```

La sentencia LOAD DATA INFILE lee registros desde un archivo de texto para ponerlos en una tabla a una velocidad muy alta. Si se especifica LOCAL, el archivo es leído desde el ordenador cliente. Si LOCAL no se especifica, el archivo debe estar ubicado en el servidor (LOCAL está disponible en la versión 3.22.6 o posterior de MySQL).

Por razones de seguridad, al leer archivos de texto localizados en el servidor, los archivos deben residir o bien en el directorio de la base de datos o ser leíbles por todos. Además, para utilizar LOAD DATA INFILE en archivos del servidor, debes tener el privilegio FILE en el servidor. Puedes ver la sección 4.2.7. [Privilegios aportados].

En MySQL 3.23.49 y MySQL 4.0.2. LOCAL sólo funcionará si no has iniciado mysqld con -–local-infile=0, o si no has activado tu cliente para soportar LOCAL. Puedes ver la sección 4.2.4 [LOAD DATA LOCAL].

Si especificas LOW_PRIORITY, la ejecución de LOAD DATA se retrasa hasta que no haya clientes leyendo de la tabla.

Si especificas CONCURRENT con una tabla MyISAM, otras llamadas pueden recuperar datos desde la tabla mientras se ejecuta LOAD DATA. Utilizando esta opción desde luego afectará a la realización del LOAD DATA un poco incluso si no existen otras llamadas la tabla en el mismo momento.

Utilizando LOCAL será algo más lento que dejar que el servidor acceda al archivo directamente, debido a que los contenidos del archivo deben viajar desde el cliente hasta el servidor. Por otro lado, no necesitas el privilegio FILE para leer archivos locales.

Si estás utilizando MySQL con versión anterior a 3.23.24 no puedes leer desde un FIFO [pila] con LOAD DATA INFILE. Si necesitas leer desde una FIFO (por ejemplo la salida desde un gunzip), utiliza a cambio LOAD DATA LOCAL INFILE.

Puedes también leer archivos de datos utilizando la utilidad mysqlimport; opera enviando el comando LOAD DATA INFILE al servidor. La opción -–local causa que mysqlimport lea archivos desde el cliente. Puedes especificar la opción --compress para una mejor ejecución sobre redes lentas si el cliente y el servidor soportan el protocolo comprimido.

Al ubicar archivos en el servidor, éste sigue las siguientes reglas:

- Si se da una ruta absoluta, el servidor lo utiliza tal cual.
- Si se da una ruta relativa con uno o más componentes, el servidor busca el archivo partiendo del directorio del servidor de la base de datos.
- Si se da un archivo sin componentes, el servidor busca el archivo en el directorio de la base de datos actual.

Nota que estas reglas significan que un archivo dado como ‘./myfile.txt’ se lee desde el directorio del servidor de la base de datos, así como el archivo ‘myfile.txt’ se lee desde el directorio de la base de datos actual. Por ejemplo, la siguiente sentencia LOAD DATA lee el archivo ‘data.txt’ desde el directorio de la base de datos para db1 porque db1 es la base de datos actual, incluso sí la sentencia carga explícitamente el archivo en una tabla de la base de datos db2:

```
mysql> USE db1;
mysql>LOAD DATA INFILE “data.txt” INTO TABLE db2.my_table;
```

Las palabras clave REPLACE e IGNORE controlan el manejo de la introducción de registros que duplican los registros existentes en valores clave únicos. Si especificas IGNORE, las filas entradas que dupliquen claves existentes serán saltadas. Si no especificas ninguna de las opciones, se da un error cuando se encuentra una duplicación de valores clave, y el resto del archivo de texto se ignora.

Si utilizas LOAD DATA INFILE en una tabla vacía MyISAM, todos los índices no únicos se crearán en un proceso por separado (como en REPAIR). Esto hace que normalmente LOAD DATA INFILE trabaje más rápido cuando tienes varios índices.

LOAD DATA INFILE es un complemento de SELECT...INTO OUTFILE. Puedes ver la sección 6.4.1. [SELECT]. Para escribir datos desde una base de datos a un archivo, utiliza SELECT....INTO OUTFILE. Para cargar de nuevo el archivo a la base de datos, utiliza LOAD DATA INFILE. La sintaxis de las cláusulas FIELDS y LINES es la misma para ambos comandos. Ambas cláusulas son opcionales, pero FIELDS debe preceder LINES si se especifican ambas.

Si especificas una cláusula FIELDS, cada una de las subcláusulas (TERMINATED BY, [OPTIONALLY] ENCLOSED BY, y ESCAPED BY) son también opcionales, excepto cuando debas especificar al menos una de ellas.

Si no especificas FIELDS, el valor por defecto sería el mismo que si hubieras escrito esto:

FIELDS TERMINATED BY '\t' ENCLOSED BY '' ESCAPED BY '\\'

Si no especificas LINES, el valor por defecto es el mismo que si hubieras escrito esto:

LINES TERMINATED BY '\n'

En otras palablas, los valores por defecto causal que LOAD DATA INFILE actúe como sigue al leer las entradas:

- Mira la longitud de la línea y los saltos de línea.
- Rompe las líneas en campos en las tabulaciones.
- No espera que los campos estén enmarcados en ningún carácter.
- Interpreta las ocurrencias de la tabulación, salto de línea, o '\' precedido por '\' como caracteres literales que son parte de los valores del campo.

De forma análoga, los valores por defecto de SELECT...INTO OUTFILE actúen como sigue en el momento de escribir una salida:

- Escribe tabulaciones entre campos.
- No encierra los campos entre caracteres.
- Utiliza '\' para escapar casos de tabulaciones, saltos de línea o '\' que tengan lugar en los valores de los campos.
- Escribe saltos de línea al final de la línea.

Nota que para escribir FIELDS ESCAPED BY '\\', debes especificar dos contrabarras para que el valor se lea como una contrabarra simple.

La opción IGNORE number LINES puede ser utilizada para ignorar los nombres de las cabeceras de columna al inicio del archivo:

mysql> LOAD DATA INFILE "/tmp/file_name" INTO TABLE test IGNORE 1 LINES;

Cuando utilizas SELECT ... INTO OUTFILE en tándem con LOAD DATA INFILE para leer y escribir posteriormente datos de una base de datos, las opciones de manejo de campo y línea para ambos comandos deben coincidir. En cualquier otro caso, LOAD DATA INFILE no interpretará propiamente los contenidos del archivo. Supón que utilizas SELECT...INTO OUTFILE para escribir un archivo con los campos delimitados por comas:

mysql>SELECT * INTO OUTFILE 'data.txt'

```
        FIELDS TERMINATED BY ‘,’
        FROM...;
```

Al leer el archivo delimitado por coma, la sentencia correcta sería:

```
mysql>LOAD DATA INFILE ‘data.txt’ INTO TABLE table2
        FIELDS TERMINATED BY ‘,’
```

Si en cambio trataras de leer el archivo con la siguiente sentencia, no trabajaría debido a que instruye LOAD DATA INFILE para localizar tabulaciones entre campos:

```
mysql> LOAD DATA INFILE ‘data.txt’ INTO TABLE table2
        FIELDS TERMINATED BY ‘\t’;
```

El resultado que parecería obtenerse es que cada línea se interpretaría como un solo campo.

LOAD DATA INFILE puede ser utilizado para leer archivos obtenidos desde orígenes externos, también. Por ejemplo, un archivo en formato dBASE tendrá los campos separados por comas y encerrados por comillas dobles. Si las líneas del archivo terminan con saltos de línea, el comando siguiente muestra las opciones de manejo de campos y líneas que utilizarías para cargar el archivo:

```
mysql> LOAD DATA INFILE ‘data.txt’ INTO TABLE table_name
        FIELDS TERMINATED BY ‘,’ ENCLOSED BY ‘”’
        LINES TERMINATED BY ‘\n’;
```

Cualquiera de las opciones de manejo de campos o línea puede especificar una cadena vacía (‘’). Si no está vacía, los valores de FIELDS [OPTIONALLY] ENCLOSED BY y FIELDS ESCAPED BY pueden ser un carácter simple. Los valores de FIELDS TERMINATED BY y LINES TERMINATED BY pueden ser de más de un carácter. Por ejemplo, para escribir líneas que terminen pares de retorno de carro-nueva línea, o para leer un archivo que contiene tales líneas, especifica LINES TERMINATED BY ‘\r\n’. Por ejemplo, para leer un archivo de bromas, que están separadas con una línea de %% dentro de una tabla SQL, puedes hacer esto:

```
CREATE TABLE jokes (a INT NOT NULL AUTO_INCREMENT PRIMARY KEY, joke TEXT NOT
NULL) ;
LOAD DATA INFILE “tmp/jokes.txt” INTO TABLE jokes FIELDS TERMINATED BY “” LINES
TERMINATED BY “\n%%\n” (joke);
```

FIELDS [OPTIONALLY] ENCLOSED BY controla los campos enmarcados. Para las salidas (SELECT...INTO OUTFILE), si omites la palabra OPTIONALLY, todos los campos serán encerrados por el carácter indicado. Un ejemplo de tal salida (utilizando una coma como delimitador de campo) , se puede ver aquí:

```
“ 1”, “a string”,”100.20”
“ 2”, “a string containing a , comma”,”102.20”
“ 3”, “a string containing a \” quote”,”102.20”
“ 4”, “a string containing a \”, quote and comma”,”102.20”
```

Si especificas OPTIONALLY, el carácter ENCLOSED BY se utilizará sólo para encerrar campos del tipo CHAR y VARCHAR:

1, “a string”,100.20
2, “a string containing a , comma”,102.20
3, “a string containing a \” quote”,102.20
4, “a string containing a \”, quote and comma”,102.20

Nota que las ocurrencias del carácter ENCLOSED BY dentro de un valor de campo se pueden saltar al utilizar el carácter de ESCAPED BY. También puedes ver que si especificas un valor vacío para ESCAPED BY, es posible generar una salida que no puede ser propiamente leída por LOAD DATA INFILE. Por ejemplo, la salida vista anteriormente podría aparecer como la siguiente si se dejara vacío el carácter de escape. Observa qe el segundo campo de la cuarta línea contiene una coma de más, lo que provoca la interpretación de fin de campo:

1, “a string”,100.20
2, “a string containing a , comma”,102.20
3, “a string containing a ” quote”,102.20
4, “a string containing a ”, quote and comma”,102.20

Para la entrada, el carácter ENCLOSED BY, si está presente, se elimina de los valores finales de los campos. (esto es verdad aunque se especifique OPTIONALLY; OPTIONALLY no tiene ningún efecto en la interpretación de entradas). Las ocurrencias del ENCLOSED BY precedidas por el ESCAPED BY se interpretan como parte del valor del campo. En adición, los caracteres duplicados de ENCLOSED BY hallados en los valores de campos se consideran como un solo carácter ENCLOSED BY si el campo en sí mismo empieza con tal carácter. Por ejemplo, si se especifica ENCLOSED BY ‘”’, las comillas se manejan como se ve a continuación:

“ The ““BIG”” boss” --> The “BIG” boss
“ The “BIG” boss” --> The “BIG” boss
The ““BIG”” boss --> The ““BIG”” boss

FIELDS ESCAPED BY controla cómo escribir o leer caracteres especiales. Si el carácter de FIELDS ESCAPED BY no está vacío, se utiliza para prefijar los siguientes caracteres en la salida:

- El carácter de FIELDS ESCAPED BY.
- El carácter de FIELDS [OPTIONALLY] ENCLOSED BY.
- El primer carácter de los valores de FIELDS TERMINATED BY y LINES TERMINATED BY.
- ASCII 0 (lo que actualmente se escribe después del carácter de escape, no un valor cero de byte).

Si el carácter de FIELDS ESCAPED BY está vacío, no se escapan caracteres. Es probable que esto no sea una buena idea, particularmente en los valores de campos en los que tus datos contengan alguno de los caracteres de la lista dada.

Para la entrada, si el carácter de FIELDS ESCAPED BY no está vacío, las ocurrencias de este carácter se eliminan y el carácter que le sigue se toma literalmente como una parte del valor del campo. Las excepciones son un ‘0’ o ‘N’ escapado (por ejemplo, \0 o \N si el carácter de escape es ‘\’). Estas secuencias se interpretan como los valores ASCII 0 (un byte de valor cero) y NULL. Puedes ver más adelante las reglas del manejo de NULL.

Para más información sobre ‘\’ sintaxis de escape, puedes ver la sección 6.1.1. [Literals].

En ciertos casos, las opciones de manejo de campos y líneas interactúan:

- Si LINES TERMINATED BY es una cadena vacía y FIELDS TERMINATED BY no está vacío, las líneas se terminan con el carácter de FIELDS TERMINATED BY.
- Si FIELDS TERMINATED BY y FIELDS ENCLOSED BY están vacíos, se utiliza el formato de ancho de columna fijo. Con este formato, no se utilizan delimitadores entre campos. En vez de eso, las columnas se escriben y leen utilizando los anchos de columna "vistos". Por ejemplo, si una columna columna se declara como INT(7), los valores de la columna se escribirán utilizando campos de 7 caracteres. En la entrada, los valores para la columna se obtienen leyendo 7 caracteres. El formato de ancho fijo afecta al manejo de los valores NULL; puedes verlo posteriormente. Nota que el formato del ancho fijo no funcionará si utilizas un juego de caracteres multi-byte.

El manejo de valores NULL varía, dependiendo de las opciones de FIELDS y LINES utilizadas:

- Para los valores por defecto de FIELDS y LINES, NULL se escribe como \N para la salida y \N se lee como NULL en la entrada (suponiendo que el carácter ESCAPED BY es '\').
- Si FIELDS ENCLOSED BY no está vacío, un campo que contenga la palabra literal NULL como valor se leerá como NULL (esto difiere de la palabra NULL encerrada en entre los caracteres de FIELDS ENCLOSED BY, que se lee como cadena 'NULL').
- Si FIELDS ESCAPED BY está vacío, NULL se escribe como palabra NULL.
- Con el formato de ancho fijo (que tiene lugar cuando FIELDS TERMINATED BY y FIELDS ENCLOSED BY están vacíos), NULL se escribe como cadena vacía. Nota que esto causa que tanto los valores NULL como las cadenas vacías en una tabla sin indistinguibles dado que ambas se escriben como cadenas vacías. Si necesitas diferenciarlas al leer de nuevo el archivo, no deberías usar el formato de ancho fijo.

Algunos casos no se soportan por LOAD DATA INFILE:
- Las filas de ancho fijo (FIELDS TERMINATED BY y FIELDS ENCLOSED BY ambos vacíos) y las columnas BLOB o TEXT.
- Si especificas un separador que es igual o es prefijo de otro, LOAD DATA INFILE es incapaz de interpretar la entrada propiamente. Por ejemplo, las siguientes cláusulas FIELDS podrían causar problemas:

  FIELDS TERMINATED BY '"' ENCLOSED BY '"'

* Si FIELDS ESCAPED BY está vacío, un valor de campo que contiene una ocurrencia de FIELDS ENCLOSED BY o LINES TERMINATED BY seguido del valor de FIELDS TERMINATED BY causará que LOAD DATA INFILE pare de leer un campo o línea antes del final. Esto sucede porque LOAD DATA INFILE no puede determinar propiamente donde termina el valor del campo o línea.

El siguiente ejemplo lee todas las columnas de la tabla persondata:

```
mysql> LOAD DATA INFILE 'persondata.txt' INTO TABLE persondata;
```

No se especifica ninguna lista de campos, con lo que LOAD DATA INFILE espera que las filas de entrada contengan un campo por cada columna. Se utilizan los valores por defecto de FIELDS y LINES.

Si deseas cargar sólo alguna de las columnas de las tablas, especifica una lista de campos:

```
mysql> LOAD DATA INFILE 'persondata.txt'
       INTO TABLE persondata (col1,col2,...);
```

Debes especificar además una lista de campos si el orden de los campos en el archivo de entrada difiere del orden de los de la tabla. En cualquier otro caso, MySQL no puede definir cómo hacer coincidir los campos de entrada con las columnas de la tabla.

Si una fila tienen muy pocos campos, las columnas para las que no se presenta ningún campo de entrada se sitúa en el valor por defecto. La asignación de valores por defecto se describe en la sección 6.5.3 [CREATE TABLE].

Un valor de campo vacío se interpreta diferente que el hecho que el valor falte:

- Para tipos cadenas, la columna toma el valor de cadena vacía.
- Para valores numéricos, la columna vale 0.
- Para tipos de hora y fecha, toma el valor apropiado de "fecha cero", según el caso. Puedes ver la sección 6.2.2. [Date and time types].

Nota que estos son los mismos valores que resulta si asignas una cadena vacía explícitamente a una columna de tipo cadena, número o fecha/hora en una sentencia INSERT o UPDATE.

Las columnas TIMESTAMP sólo se sitúan al valor actual si hay un valor NULL, o (sólo para la primera columna TIMESTAMP) si la columna TIMESTAMP se deja fuera de la lista de campos cuando esta lista se especifica.

Si una fila de entrada tiene muchos archivos, los campos extra se ignoran y el número de alertas se incrementa.

LOAD DATA INFILE supone todas las entradas como cadenas, por lo que no puedes utilizar valores numéricos para las columnas ENUM o SET del mismo modo que podrías hacerlo con INSERT. Todos los valores ENUM y SET deben especificarse como cadenas!

Si estás utilizando la API de C, puedes obtener información sobre la consulta llamando a la función mysql_info() cuando la consulta LOAD DATA INFILE finalice. El formato de la cadena de información se puede ver aquí:

Records: 1 Deleted: 0 Skipped: 0 Warnings: 0

Las alertas tienen lugar bajo las mismas circunstancias en las que tendrían lugar al utilizar la sentencia INSERT (puedes ver la sección 6.4.3 [INSERT]), escepto que LOAD DATA INFILE también genera alertas cuando hay demasiados pocos campos en la fila de entrada. Las alertas no se almacenan en ninguna parte; el número de alertas sólo se puede utilizar como un indicador de si todo fue bien. Si obtienes alertas y quieres saber exactamente por qué se dieron, una forma de hacerlo es utilizando SELECT...INTO OUTFILE en otro archivo y compararlo al archivo original.

Si necesitas que LOAD DATA lea desde una pila, puedes utilizar el siguiente truco:

```
mkfifo /mysql/db/x/x
chmod 666 /mysql/db/x/x
cat < /dev/tcp/10.1.1.12/4711 > /nt/mysql/db/x/x
mysql -e "LOAD DATA INFILE 'x' INTO TABLE x" x
```

Si estás utilizando una versión de MySQL anterior a 3.23.35, sólo puedes hacer lo anterior con LOAD DATA LOCAL INFILE.

Para más información sobre la eficiencia de INSERT respecto a LOAD DATA INFILE y la velocidad de LOAD DATA INFILE, puedes ver la sección 5.2.9 [Insert speed].

6.4.10. Sintaxis DO

DO expression, [expression...]

Ejecuta la expresión pero no retorna resultados. Esta es una abreviatura de SELECT expression, expression..., pero tiene la ventaja de que es ligeramente más rápido cuando no te preocupa el resultado.

Principalmente es útil con las funciones que tiene efectos laterales, como RELEASE_LOCK.

6.5. Definición de datos: CREATE, DROP, ALTER

6.5.1. Sintaxis CREATE DATABASE

CREATE DATABASE [IF NOT EXISTS] db_name

CREATE DATABASE crea una base de datos con el nombre dado. Las reglas para los nombres de base de datos permitidos se dan en la sección 6.1.2. [Legal names]. Se da un error si la base de datos existe y no se ha especificado IF NOT EXISTS.

Las bases de datos en MySQL se implementan como directorios que contienen los archivos que corresponden a la base de datos. Dado que no hay tablas en la base de datos cuando ésta se crea, la sentencia CREATE DATABASE sólo crea un directorio bajo el directorio de datos de MySQL.

Puedes crear también bases de datos con mysqladmin. Puedes ver la sección 4.8 [Client-side scripts].

6.5.2. Sintaxis DROP DATABASE

DROP DATABASE [IF EXISTS] db_name

DROP DATABASE elimina las tablas de la base de datos y la propia base de datos. Si realizas un DROP DATABASE sobre el enlace simbólico de una base de datos, tanto la base de datos como el enlace se eliminan. **Ten mucho cuidado con este comando!**

DROP DATABASE retorna el número de archivos que se eliminaron del directorio de la base de datos, que normalmente es el triple del número de tablas de la base de datos, ya que cada tabla se corresponde con un archivo '.MYD', un archivo '.MYI' y un archivo '.frm'.

El comando DROP DATABASE elimina del directorio de la base de datos todos los archivos con las siguientes extensiones:

.BAK, .DAT, .HSH, .ISD, .ISM, .MRG, .MYD, MYI, .db,.frm

Todos los subdirectorios que consisten en 2 dígitos (directorios RAID) se eliminan también.

En la versión 3.22 de MySQL o posterior, puedes utilizar las palabras clave IF EXISTS para prevenir un error en caso que la base de datos no exista.

Puedes también eliminar bases de datos com mysqladmin. Puedes ver la sección 4.8. [Client-side scripts].

6.5.3. Sintaxis CREATE TABLE

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] table_name [(create_definition,...)]
[table_options] [select_statement]

Definición de creación (create_definition):

col_name type [NOT NULL |NULL] [DEFAULT default_value] [AUTO_INCREMENT]
[PRIMARY KEY] [reference_definition]

- PRIMARY KEY (index_col_name,....)
- KEY [index_name] (index_col_name,...)
- INDEX [index_name] (index_col_name...)
- UNIQUE [INDEX] [index_name] (index_col_name,...)
- FULLTEXT [INDEX] [index_name] (index_col_name,...)
- [CONSTRAINT symbol] FOREIGN KEY [index_name] (index_col_name,...)
  [reference_definition]
- CHECK (expr)

tipos (type):

TINYINT[(length)] [UNSIGNED] [ZEROFILL]

- SMALLINT[(length)] [UNSIGNED] [ZEROFILL]
- MEDIUMINT[(length)] [UNSIGNED] [ZEROFILL]
- INT[(length)] [UNSIGNED] [ZEROFILL]
- INTEGER[(length)] [UNSIGNED] [ZEROFILL]
- BIGINT[(length)] [UNSIGNED] [ZEROFILL]
- REAL[(length,decimals)] [UNSIGNED] [ZEROFILL]
- DOUBLE[(length,decimals)] [UNSIGNED] [ZEROFILL]
- FLOAT[(length,decimals)] [UNSIGNED] [ZEROFILL]
- DECIMAL[(length,decimals)] [UNSIGNED] [ZEROFILL]
- NUMERIC[(length,decimals)] [UNSIGNED] [ZEROFILL]
- CHAR[(length)] [BINARY]
- VARCHAR(length) [BINARY]
- DATE
- TIME
- TIMESTAMP
- DATETIME
- TINYBLOB
- BLOB
- MEDIUMBLOB
- LONGBLOB
- TINYTEXT
- TEXT
- MEDIUMTEXT
- LONGTEXT
- ENUM(value1,value2,value3,...)
- SET(value1,value2,value3,...)

index_col_name:

col_name[(lenght)]

reference_definition:

REFERENCES table_name[(index_col_name,...)]
[MATCH FULL | MATCH PARCIAL]
[ON DELETE reference_option]
[ON UPDATE reference_option]

reference_option:

RESTRICT | CASCADE | SET NULL | NO ACTION |SET DEFAULT

table_options:

TYPE={BDB | HEAP | ISAM | InnoDB | MERGE | MRG_MYISAM | MYISAM}
- AUTO_INCREMENT = #
- AVG_ROW_LENGTH = #
- CHECKSUM={0 | 1}
- COMMENT = “string”
- MAX_ROWS = #
- MIN_ROWS = #
- PACK_KEYS={0|1|DEFAULT}
- PASSWORD = “string”
- DELAY_KEY_WRITE={0|1}
- ROW_FORMAT={default|dynamic|fixed|compressed}
- RAID_TYPE={1|STRIPPED|RAIDO} RAID_CHUNKS=# RAID_CHUNKSIZE=#
- UNION = {table_name,[table_name...]}
- INSERT_METHOD={NO|FIRST|LAST}
- DATA DIRECTORY = “absolute path to directory”
- INDEX DIRECTORY = “absolute path to directory”

select_statement:

[IGNORE |REPLACE] SELECT ... (alguna sentencia SELECT legal)

CREATE TABLE crea una tabla con un nombre dado en la base de datos actual. Las reglas para los nombres permitidos de tabla vienen dadas en la sección 6.1.2 [Legal names]. Se da un error si no existe base de datos actual o si la tabla ya existe.

En MySQL 3.22 o posterior, el nombre de la tabla puede ser especificado como db_name.table_name. Esto funciona independientemente de si existe una base de datos actual.

En MySQL 3.23, puedes utilizar la palabra clave TEMPORARY cuando creas una tabla. Una tabla temporal se borrará automáticamente si se pierde una conexión y el nombre es por conexión. Esto significa que dos conexiones diferentes pueden utilizar el mismo nombre de tabla temporal sin entrar en conflicto con la otra o con una tabla existente del mismo nombre. (la tabla existente estará oculta hasta que la tabla temporal se borre). En MySQL 4.0.2. el usuario debe disponer del privilegio CREATE TEMPORARY TABLES para poder crear tablas temporales.

En MySQL 3.23 o superior, puedes utilizar las palabras clave IF NOT EXISTS para que no tenga lugar un error en caso que la tabla exista. Nota que no hay verificación de que las estructuras de las tablas sean idénticas.

Cada tabla table_name se representa con algunos archivos en el directorio de la base de datos. En el caso de las tablas tipo MyISAM obtendrás:

Tabla	Finalidad
table_name.frm	Archivo de definición de la tabla (formato)
table_name.MYD	Archivo de datos
table_name.MYI	Archivo de índice

Para más información sobre las propiedades de los diferentes tipos de columnas, puedes ver la sección 6.2 [Column types].

- Si no se especifica ni NULL ni NOT NULL, la columna se trata como si se hubiera especificado NULL.
- Una columna de valores enteros puede tener el atributo adicional AUTO_INCREMENT. Cuando insertas el valor NULL (recomendado) o 0 en una columna AUTO_INCREMENT, el valor se sitúa en value+1, donde value es el valor mayor actual para la columna dada. Las secuencias de AUTO_INCREMENT se inician en 1. Puedes ver la sección 8.4.3.30 [mysql_insert_id()].

  Si borras la fila que contiene el valor máximo para la columna AUTO_INCREMENT, el valor se reutilizará con una tabla ISAM o BDB, pero no con una MyISAM o InnoDB. Si borras todos los registros de la tabla con DELETE FROM table_name (sin WHERE) en modo AUTOCOMMIT, la secuencia se reinicia para todas las tablas.

  Nota: sólo puede haver una columna AUTO_INCREMENT para cada tabla, y debe ser indexada. MySQL 3.23 trabajará también adecuadamente si la columna AUTO_INCREMENT sólo tiene valores positivos. Insertar un valor negativo se trata como si se insertara un valor positivo muy grande. Esto se hace para evitar los problemas de precisión cuando los números 'saltan' de positivo a negativo y también para asegurar que no tiene lugar una columna AUTO_INCREMENT que contenga el 0.

  En las tablas MyISAM y BDB puedes especificar una columna secundaria AUTO_INCREMENT en una clave multi-columna. Puedes ver la sección 3.5.9. [example-AUTO_INCREMENT].

  Para compatibilizar MySQL con aplicaciones ODBC, puedes encontrar la última columna insertada con la siguiente consulta:

  SELECT * FROM table_name WHERE auto_col IS NULL

- Los valores NULL se manejan de forma diferente para las columnas TIMESTAMP que para el resto. No puedes almacenar un NULL literal en una columna TIMESTAMP; dando valor NULL a la columna, se le da la fecha y hora actuales. Daddo que las columnas TIMESTAMP funcionan de esta manera, los valores NUT y NOT NULL no se aplican de la forma normal y se ignoran si los especificas.

  Por otro lado, para simplificar el uso de columnas TIMESTAMP en clientes MySQL, el servidor reporta que a tales columnas se les pueden asignar valores NULL (que es cierto),

aunque TIMESTAMP nunca pueda contener valores NULL actualmente. Puedes verlo cuando utilizas DESCRIBE table_name para obtener la descripción de tu tabla.

Nota que dar valor 0 a una columna TIMESTAMP no es lo mismo que darle valor NULL, dado que 0 es un valor de tiempo válido.

- Un valor DEFAULT debe ser una constante, no puede ser una función o expresión. Si no se especifica un valor por defecto para una columna, MySQL automáticamente le asigna uno. Si la columna puede tomar NULL como valor, el valor por defecto es NULL. Si la columna se declara como NOT NULL, el valor por defecto depende del tipo de columna:
    - Para valores numéricos diferentes de los declarados con el atributo AUTO_INCREMENT, el valor por defecto es 0. Para una columna AUTO_INCREMENT, el valor por defecto es el siguiente de la secuencia.
    - Para tipos de fecha y hora diferentes de TIMESTAMP, el valor por defecto es el cero apropiado para el tipo. Para la primera columna TIMESTAMP de la tabla, el valor por defecto es el valor actual de fecha y hora. Puedes ver la sección 6.2.2. [Date and time types].
    - Para tipos cadena diferentes de ENUM, el valor por defecto es una cadena vacía. Para ENUM, el valor por defecto es el primer valor de la enumeración (Si no has especificado explícitamente otro valor por defecto en la directiva DEFAULT).

  Los valores por defecto deben ser constantes. Esto significa, por ejemplo, que no puede ser dado por defecto el valor de una función como NOW() o CURRENT_DATE.

- KEY es un sinónimo de INDEX.
- En MySQL, una clave UNIQUE sólo puede tener valores diferentes. Se da un error si tratas de añadir una nueva fila con un valor clave que concuerda con otro existente.
- Una PRIMARY KEY es una UNIQUE con la restricción añadida de que todas las columnas clave deben definirse como NOT NULL. En MySQL, la clave se llama PRIMARY. Una tabla sólo puede tener una PRIMARY KEY. Si no tienes una PRIMARY KEY y algunas aplicaciones la piden, MySQL retornará la primera clave UNIQUE, que no tenga valores nulos, como la PRIMARY KEY.
- Una PRIMARY KEY puede ser un índice multicolumna. Sin embargo, no puedes crear un índice multicolumna utilizando el atributo clave PRIMARY KEY en la especificación de una columna. Haciéndolo marcará sólo esta columna como primaria. Debes utilizar la sintaxis PRIMARY KEY (index_col_name,...).
- Si la clave PRIMARY o UNIQUE consiste en sólo una columna y es de tipo entero, te puedes referir a esta como _rowid (a partir de la versión 3.23.11).
- Si no asignas un nombre a un índice, se asignará al índice el mismo nombre que el primer index_col_name, con un sufijo opcional (_2,_3,...) para que sea único. Puedes ver los nombres de los índices para una tabla utilizando SHOW INDEX FROM table_name. Puedes ver la sección 4.5.6 [SHOW].
- Sólo las tablas de tipo MyISAM, InnoDB, y BDB soportan los índices en columnas que pueden tener valores nulos. En otros casos debes declarar tales columnas como NOT NULL o se da un error.
- Con la sintaxis col_name(length), puedes especificar un índice que utiliza sólo una parte de la columna CHAR o VARCHAR. Esto puede hacer que el archivo de índices sea mucho más pequeño. Puedes ver la sección 5.4.4. [Indexes].
- Sólo el tipo de tabla MyISAM soporta la indexación en columnas de tipo BLOB o TEXT. Cuando se inserta un índice en una columna BLOB o TEXT, DEBES especificar siempre la longitud del índice:

```
CREATE TABLE test (blob_col BLOB, INDEX (blob_col(10)));
```

- Cuando utilizas ORDER BY o GROUP BY con una columna BLOB o TEXT, sólo se utilizan los primeros max_sort_length bytes. Puedes ver la sección 6.2.3.2. [BLOB].
- En la versión 3.23.23 o posterior, puedes crear también índices especiales FULLTEXT. Se utilizan para la búsqueda en textos completos. Los índices FULLTEXT sólo es soportado por el tipo MyISAM. Sólo pueden ser creados desde columnas VARCHAR o TEXT. La indexaciín siempre tiene lugar sobre la columna entera, ya que la indexación parcial no se soporta. Puedes ver la sección 6.8. [FULLTEXT] para los detalles de la operación.
- En la versiín 3.23.44 o posterior, las tablas InnoDB soportan el chequeo de las restricciones de la foreign key. Puedes ver la sección 7.5 [InnoDB]. Para otros tipos de tablas, el servidor MySQL interpreta la sintaxis de la FOREIGN KEY, CHECK, y REFERENCES en los comandos CREATE TABLE, pero sin que se tome una acción posterior. Puedes ver la sección 1.7.4.5. [ANSI diff foreign Keys].
- Cada columna NULL toma un bit extra, redondeado por encima al byte más cercano.
- la máxima longitud de registro en bytes puede ser calculada como sigue:

  longitud de registro= 1
  + (suma de longitudes de columnas)
  + (número de columnas NULL+7)/8
  + (número de columnas con longitud variable)

- Las opciones table_options y SELECT sólo se implementan en la versión MySQL 3.23 y posteriores. Los diferentes tipos de tablas son:

Tipo de tabla	Descripción
BDB o BerkeleyDB	Tablas a prueba de transacciones con el bloqueo de páginas. Puedes ver la sección 7.6. [BDB]
HEAP	Los datos para esta tabla sólo se almacenan en memoria. Puedes ver la sección 7.4. [HEAP]
ISAM	El manejador original de la tabla. Puedes ver la sección 7.3 [ISAM]
InnoDB	Tablas a prueba de transacciones con bloqueo de registros. Puedes ver la sección 7.5. [InnoDB]
MERGE	Una colección de tablas MyISAM utilizadas como una tabla. Puedes ver la sección 7.2. [MERGE]
MRG_MyISAM	Un alias para las tablas MERGE
MyISAM	El nuevo manejador de la tabla binaria que está sustituyendo ISAM. Puedes ver la sección 7.1. [MyISAM].

Puedes ver el capítulo 7 [Table types].

Si se especifica un tipo de tabla, y este tipo en particular no está disponible, MySQL escogerá el tipo más cercano al que has elegido. Por ejemplo, si TYPE=BDB se especifica, y esa distribución de MySQL no soporta tablas BDB, esta será creada con MyISAM.

Las otras opciones de tabla se utilizan para optimizar el comportamiento de la tabla. En varios casos, no debes especificar cualquiera de ellos. Las opciones trabajan para todos los tipos de tablas, si no se indica de otro modo:

Opción	Descripción
AUTO_INCREMENT	El siguiente valor AUTO_INCREMENT que quieres poner en tu tabla (MyISAM)
AVG_ROW_LENGTH	Una estimación de la longitud media del registro de tu tabla. Sólo necesitas poner esto para tablas grandes con registros de tamaño variable.
CHECKSUM	Dale valor 1 si quieres que MySQL mantenga una suma de control (checksum) para todos los registros (hace que la tabla se algo más lenta de actualizar, pero simplifica la localización de tablas corruptas) (MyISAM).
COMMENT	Comentario de 60 caracteres sobre la tabla
MAX_ROWS	Máximo número de registros que planeas almacenar en la tabla
MIN_ROWS	Mínimo número de registros que planeas almacenar en la tabla
PACK_KEYS	Dale valor 1 si quieres tener un índice más pequeño. Esto usualmente hace actualizaciones más lento y lo hace más rápido (MyISAM, ISAM). Dándole valor 0 desactivará el empaquetamiento de claves. Dándole el valor DEFAULT (MySQL 4.0) explicará al manejador que sólo empaquete las columnas largas de tipo CHAR/VARCHAR.
PASSWORD	Encripta el archivo '.frm' con una contraseña. Esta opción no hace nada en la versión estándar de MySQL.
DELAY_KEY_WRITE	Dale valor 1 para retrasar las actualizaciones de las claves de las tablas hasta que se cierre la tabla.
ROW_FORMAT	Define cómo los registros deberían ser almacenadas. Habitualmente esta opción sólo funciona con las tablas MyISAM, que soporta los formatos de registros DYNAMIC y FIXED. Puedes ver la sección 7.1.2. [MyISAM table formats]

Cuando utilizas una tabla de tipo MyISAM, MySQL utiliza el producto de max_rows * avg_row_length para decidir cómo de grande será la tabla resultante. Si no especificas ninguna de las opciones anteriores, el tamaño máximo de la tabla será de 4 Gigabytes (2 GB si tu sistema operativo sólo soporta tablas de este tamaño como máximo). La razón de este límite es sólo mantener en valores bajos el índice, lo que lo hace más rápido si realmente no necesitas tablas de estos tamaños.

Si no utilizas la opción PACK_KEYS, el valor por defecto es que sólo se empaquetan las cadenas, no los números. Si das PACK_KEYS=1, los números también se empaquetarán.

Al empaquetar claves de valores binarios, MySQL utilizará la compresión del prefijo. Esto significa que sólo se obtendrá un beneficio de esto si tienes varios números idénticos. La compresión del prefijo significa que cada clave necesita un byte extra para indicar cuántos bytes de la clave anterior son idénticos para la siguiente clave (nota que el puntero del registro se almacena directamente en high-byte-first-order después de la clave, para mejorar la compresión). Esto significa que si tienes varias claves iguales en dos campos de un registro, todas las siguientes 'iguales' tomarán usualmente 2 bytes (incluyendo el puntero de la fila). Compara esto a el caso ordinario en el que las siguientes claves van a tomar storage_size_for_key + pointer_size (usualmente 4). Por otro lado, si todas las claves son totalmente diferentes, vas a perder 1 byte por clave, si la clave no tiene valores NULL (en este caso la longitud de la clave empaquetada se almacenará en el mismo byte utilizado para marcar si la clave es NULL).

- Si especificas un SELECT después de una sentencia CREATE, MySQL creará nuevos campos para todos los elementos en el SELECT. Por ejemplo:

```
mysql> CREATE TABLE test (a INT NOT NULL AUTO_INCREMENT)
                    PRIMARY KEY (a), KEY (b)
                    TYPE=MyISAM SELECT b,c FROM test2;
```

Esto creará una tabla MyISAM con tres columnas, a, b y c. Las columnas de la sentencia SELECT se adjuntan a la derecha de la tabla, no solapada en esta. Toma el siguiente ejemplo:

```
mysql> SELECT * FROM foo;
```

n
1

```
mysql> CREATE TABLE bar (m INT) SELECT n FROM foo;

Query OK, 1 row affected (0.02 sec)
Records: 1   Duplicates: 0   Warnings: 0

mysql>SELECT * FROM bar;
```

m	n
NULL	1

Para cada fila de la tabla foo, se inserta una fila en bar con los valores de foo y los valores por defecto de las nuevas columnas.

CREATE TABLE…SELECT no creará automáticamente índices por ti. Esto se hace intencionadamente para hacer que el comando sea tan flexible como sea posible. Si quieres tener índices en la creación de tabla, deberías especificarlos antes de la sentencia SELECT:

```
mysql> CREATE TABLE bar (UNIQUE (n)) SELECT n FROM foo;
```

Si tiene lugar cualquier error al copiar los datos a la tabla, serán borrados automáticamente.

Para asegurarse que el log de actualización y el log binario pueden ser usados para recrear las tablas originales, MySQL no permitirá inserciones concurrentes durante el CREATE TABLE…SELECT.

- La opción RAID_TYPE te ayudará a romper el límite 2G/4G para los archivos de datos MyISAM (no el archivo de índice) en sistemas operativos que no soportan grandes archivos. Nota que esta opción no se recomienda para sistemas de archivos que soportan grandes archivos!

  Puedes obtener más velocidad en el cuello de botella de I/O creando directorios RAID en diferentes discos fijos. RAID_TYPE trabajará en cualquier Sistema Operativo, a partir de

que hayas configurado MySQL con --with-raid. Por ahora el único RAID_TYPE permitido es STRIPPED (1 y RAIDO son alias de éste).

Si especificas RAID_TYPE=STRIPPED para la tabla MyISAM, MyISAM creará subdirectorios RAID_CHUNKS llamados 00, 01, 02 en directorio de la base de datos. En cada uno de estos directorios MyISAM creará una table_name.MYD. Al escribir datos en el archivo, el manejador del RAID mapeará los primeros RAID_CHUNKSIZE*1024 bytes del primer archivo, los siguientes RAID_CHUNKSIZE*1024 bytes del siguiente archivo, y así hasta el final.

- UNION se utiliza cuando quieres utilizar una colección de tablas idénticas como una única. Esto sólo funciona con tablas MERGE. Puedes ver la sección 7.2 [MERGE].

  Por el momento necesitas tener los privilegios de SELECT, UPDATE y DELETE en las tablas que mapees para una tabla MERGE. Todas las tablas mapeadas deben estar en la misma base de datos que la tabla MERGE.

- Si quieres insertar datos en un archivo MERGE, debes especificarlo con INSERT_METHOD dentro de la tabla en la que el registro se insertaría. Puedes ver la sección 7.2 [MERGE]. Esta opción se introdujo en la versión 4.0.0. de MySQL.
- En la tabla creada la clave PRIMARY se ubicará primero, seguido por todas las claves UNIQUE y luego las claves normales. Esto ayuda al optimizador de MySQL a prioritizar qué clave utilizar y así detectar más rápidamente los valores duplicados de las claves UNIQUE.
- Utilizando DATA DIRECTORY="directorio" o INDEX DIRECTORY="directory" puedes especificar donde debería poner el manejador de la tabla sus archivos de índices. Nota que el directorio debería ser una ruta completa del directorio (no relativo).

  Ésto sólo funciona para las tablas MyISAM en MySQL 4.0, cuando no estás utilizando la opción --skip-symlink. Puedes ver la sección 5.6.1.2 [Symbolic links to tables].

6.5.3.1. Especificación de cambios silenciosos [Silent changes] de columnas

En algunos casos, MySQL cambia sin indicarlo una especificación de columna a partir de la dada en la sentencia CREATE TABLE (esto puede ocurrir con ALTER TABLE):

- Columnas VARCHAR con una longitud inferior a cuatro se convierten a CHAR.
- Si alguna columna de una tabla tiene una longitud variable, el registro entero tiene longitud variable como consecuencia. Por lo tanto, si una tabla contiene cualquier columna de longitud variable (VARCHAR, TEXT o BLOB), todas las columnas CHAR superiores a tres caracteres se cambian a columnas VARCHAR. Esto no en ningún sentido afecta a cómo utilices las columnas; en MySQL, VARCHAR es sólo una forma diferente para almacenar caracteres. MySQL realiza la conversión porque ahorra espacio y agiliza las operaciones de tabla. Puedes ver el capítulo 7 [Table types].
- Los tamaós de visualización de TIMESTAMP deben ser pares y dentro del rango de 2 a 14. Si especificas un tamaño de visualización de 0 o mayor que 14, el tamaño se ajusta a 14. Los tamaños de longitud impar en el rango de 1 a 13 se ajustan al valor par superior.
- No puedes almacenar un valor NULL literal en un campo TIMESTAMP; dándole valor nulo se actualizará a la fecha y hora actuales. Dado que las columnas TIMESTAMP trabajan de esta forma, los atributos NULL y NOT NULL no se aplican normalmente y son ignorados si los especificas. DESCRIBE table_name siempre reporta que a una columna TIMESTAMP se le puede asignar un valor NULL.

- MySQL convierte ciertos tipos de columnas utilizados por otros tipos de bases de datos SQL a los tipos MySQL. Puedes ver la sección 6.2.5. [Other-vendor column types].

Si quieres ver en qué casos MySQL utiliza un tipo de columna diferente al que has especificado, utiliza la sentencia DESCRIBE table_name después de crear o modificar la estructura de la tabla.

Otros cambios en diferentes tipos de columnas ocurren si comprimes una tabla utilizando myisampack. Puedes ver la sección 7.1.2.3 [Compressed Format].

6.5.4. Sintaxis de ALTER TABLE

ALTER [IGNORE] TABLE table_name alter_spec [, alter_spec...]

alter_specification:

ADD [COLUMN] cretae_definition [FIRST|AFTER column_name]
ó ADD [COLUMN] (create_definition,create_definition,…)
ó ADD INDEX [index_name] (index_col_name,…)
ó ADD PRIMARY KEY (index_col_name,…)
ó ADD UNIQUE [index_name] (index_col_name,…)
ó ADD FULLTEXT [index_name] (index_col_name,…)
ó ADD [CONSTRAINT symbol] FOREIGN KEY index_name (index_col_name,…) [reference_definition]
ó ALTER [COLUMN] col_name {SET DEFAULT literal | DROP DEFAULT}
ó CHANGE [COLUMN] old_col_name, create_definition
[FIRST | AFTER column_name]
ó MODIFY [COLUMN] create_definition [FIRST | AFTER column_name]
ó DROP [COLUMN] col_name
ó DROP PRIMARY KEY
ó DROP INDEX index_name
ó DISABLE KEYS
ó ENABLE KEYS
ó RENAME [TO] new_table_name
ó ORDER BY col
ó table_options

ALTER TABLE te permite cambiar la estructura de una tabla existente. Por ejemplo, puedes añadir o eliminar columna, crear o destruir índices, cambiar el tipo de las columnas existentes, o renombrar columnas o de la tabla en sí misma. También puedes cambiar el comentario de la tabla y el tipo de tabla. Puedes ver la sección 6.5.3. [CREATE TABLE].

Si utilizas ALTER TABLE para cambiar la especificación de columna pero DESCRIBE table_name indica que tu columna no ha cambiado, es posible que MySQL ignore tu modificación por una de las razones descritas en la sección 6.5.3.1. [Silent column changes]. Por ejemplo, si tratas de cambiar una columna VARCHAR a CHAR, MySQL utilizará aún VARCHAR si la tabla contiene otras columnas de longitud variable.

ALTER TABLE trabaja haciendo una copia temporal de la tabla original. La alteración se realiza en la copia, entonces el original se borra y la copia se renombra. Esto se hace de una manera en la que todas las actualizaciones se redirigen automáticamente a la nueva tabla sin todas las actualizaciones fallidas. Mentras ALTER TABLE se está ejecutando, la tabla original es leíble por otros clientes. Las actualizaciones y escrituras se paran hasta que la nueva tabla esté lista.

Nota que si utilizas cualquier otra opción de ALTER TABLE que no sea RENAME, M;ySQL siempre creará una tabla temporal, aunque los datos no hubieran de ser necesariamente copiados (como cuando cambias el nombre de una columna). Planeamos de mejorar esto en el futuro, pero dado que no se acostumbra a hacer ALTER TABLE, no se encuentra entre nuestras tareas más inmediatas. Para las tablas MyISAM, puedes acelerar la recreación de la parte del índice (que es la más lenta del proceso de recreación) dando un valor alto a la variable myisam_sort_buffer_size.

- Para utilizar ALTER TABLE, necesitas los privilegios de ALTER, INSERT y CREATE sobre la tabla.
- IGNORE es una extensión MySQL sobre el SQL92. Controla cómo trabaja ALTER TABLE si hay duplicados en claves únicas en la tabla. Si IGNORE no está especificado, la copia se aborta y se retorna. Si se especifica IGNORE, para los registros con duplicados en una clave única, sólo se utiliza la primera fila; las otras son borradas.
- Puedes incluír múltiples cláusulas ADD, ALTER, DROP y CHANGE en una sola sentencia ALTER TABLE. Esta es una extensión de MySQL sobre el SQL92, que sólo permite una cláusula de cada por sentencia ALTER TABLE.
- CHANGE col_name, DROP col_name y DROP INDEX son extensiones MySQL al SQL92.
- MODIFY es una extensión de Oracle al ALTER TABLE.
- La palabra opcional COLUMN es puro ruido y puede ser omitida.
- Si utilizas ALTER TABLE table_name RENAME TO new_name sin otras opciones, MySQL simplemente renombra los archivos que se corresponden a la tabla table_name. No hay necesidad de crear tabla temporal. Puedes ver la sección 6.5.5. [RENAME TABLE].
- Las cláusulas create_definition utilizan la misma sintaxis para ADD y CHANGE que en CREATE TABLE. Nota que esta sintaxis incluye el nombre de la columna, no sólo el tipo de columna. Puedes ver la sección 6.5.3. [CREATE TABLE].
- Puedes renombrar una columna utilizando la cláusula CHANGE old_col_name create_definition Para hacerlo, especifica el nombre antiguo y nuevo de la columna y el tipo que la columna tiene actualmente. Por ejemplo, para renombrar una columna INTEGER de a hacia b, puedes hacer esto:

  ```
  mysql> ALTER TABLE t1 CHANGE a b INTEGER;
  ```

  Si quieres cambiar un tipo de columna pero no el nombre, la sintaxis de CHANGE aún requiere dos nombres de columna aunque sean el mismo. Por ejemplo:

  ```
  mysql> ALTER TABLE t1 CHANGE b b BIGINT NOT NULL;
  ```

  Sin embargo, desde la versión 3.23.16a de MySQL, también puedes utilizar MODIFY para cambiar el tipo de columna sin cambiar el nombre:

  ```
  mysql> ALTER TABLE t1 MODIFY b BIGINT NOT NULL;
  ```

- Si utilizas CHANGE o NODIFY para recortar una columna para la que existe un índice en parte de la columna (por ejemplo, si tienes un índice sobre los primeros 10 caracteres de una columna VARCHAR), no puedes acortarla más que el número de caracteres indexados.
- Cuando cambias un tipo de columna utilizando CHANGE o MODIFY, MySQL trata de convertir los datos al nuevo tipo tan bien como le es posible.
- En MySQL 3.22 o posterior, puedes utilizar FIRTS o ADD...AFTER col_name para añadir una columna en la posición específica de un registro de una tabla. La acción por defecto es añadir la columna al final. Desde la versión 4.0.1. de MySQL, también puedes utilizar la palabras clave FIRST y ADD en CHANGE o MODIFY.

- ALTER COLUMN especificas un nuevo valor por defecto para una columna o borra el antiguo valor por defecto. Si el antiguo valor pord defecto es borrado y la columna puede tener valores NULL, el nuevo valor por defecto es NULL. Si la columna no puede tener tal valor, MySQL asigna un valor por defecto, tal como se describe en la sección 6.5.3. [CREATE TABLE].
- DROP INDEX elimina un índice. Esta es una extensión MySQL para el ANSI SQL92. Puedes ver la sección 6.5.8 [DROP INDEX].
- Si las columnas se borran de la tabla, también serán borradas de cualquier índice en el que formen parte. Si todas las columnas que confeccionan un índice son borradas, también se borra el índice.
- Si una tabla contiene sólo una columna, ésta no puede ser borrada. Si de lo que se trata es de borrar la tabla, utiliza a cambio DROP TABLE.
- DROP PRIMARY KEY elimina la clave primaria. Si no existe tal índice, elimina el primer índice UNIQUE de la tabla. (MySQL marca la primera clave UNIQUE como PRIMARY KEY si no existe ninguna clave primara especificada explícitamente).

  Si añades una UNIQUE INDEX a una PRIMARY KEY a una tabla, ésta se almacena antes de cualquier índice UNIQUE, por lo que MySQL puede detectar claves duplicadas tan pronto como es posible.

- ORDER BY te permite crear la nueva tabla con registros en un orden específico. Nota que la tabla no mantendrá este orden en las inserciones y eliminaciones posteriores. En algunos casos, esto puede hacer la ordenación más sencilla para MySQL si la tabla está en orden por la columna que deseas ordenar. Esta opción es útil principalmente cuando sabes que habitualmente consultarás los registros en un cierto orden; utilizando esta opción después de grandes cambios en la tabla, puedes realizar acciones más eficientes.
- Si utilizas ALTER TABLE en una tabla MyISAM, todos los índices no-únicos se crean en un proceso separado (como en REPAIR). Esto debería hacer que ALTER TABLE se agilizara cuando tienes varios índices.
- Desde MySQL 4.0. la anterior funcionalidad puede ser activada explícitamente. ALTER TABLE…DISABLE KEYS hace que MySQL pare de actualizarse índices no únicos para una tabla MyISAM. ALTER TABLE…ENABLE KEYS entonces debería ser utilizada para recrear índices que falten. Dado que MySQL lo realiza con un algoritmo especial mucho más rápido insertando claves una por una, la desactivación de claves podría dar una aceleración considerable en inserciones de bulto.
- Con la función mysql_info() de la API de C, puedes ver cuántos registros fueron copiados, y (cuando IGNORE es utilizado) cuántos registros fuero borrados por la duplicación de valores de claves únicas.
- Las cláusulas FOREIGN KEY, CHECK y REFERENCES no hacen nada actualmente. Su sintaxis se provee sólo por compatibilidad, para simplificar la portabilidad de código desde otros servidores SQL y para arrancar aplicaciones que crean tablas con referencias. Puedes ver la sección 1.7.4 [Differences from ANSI].

Aquí tenemos un ejemplo que enseña algunos usos de ALTER TABLE. Empezaremos con una tabla que se crea así:

```
mysql> CREATE TABLE t1 (a INTEGER, b CHAR(10));
```

Para renombrar la tabla de t1 a t2:

```
mysql> ALTER TABLE t1 RENAME t2;
```

Para cambiar la columna a de INTEGER a TINYINT NOT NULL (dejando el mismo nombre) y para cambiar la columna b de CHAR(10) a CHAR(20), renombrándola además de b a c:

mysql> ALTER TABLE t2 MODIFY a TINYINT NOT NULL, CHANGE b c CHAR(20);

Para añadir un nuevo campo TIMESTAMP llamada d;

mysql> ALTER TABLE t2 ADD d TIMESTAMP;

Para añadir un índice sobre la columna d, y hacer que la columna a sea primary key:

mysql> ALTER TABLE t2 ADD INDEX (d), ADD PRIMARY KEY (a);

Para eliminar la columna c:

mysql> ALTER TABLE t2 DROP COLUMN c;

Para añadir un nuevo entero AUTO_INCREMENT llamado c:

mysql> ALTER TABLE t2 ADD c INT UNSIGNED NOT NULL AUTO_INCREMENT, ADD INDEX (c)

Nota que hemos indexado c, dado que las columnas AUTO_INCREMENT deben ser indexadas, y también que declaramos c como NOT NULL, dado que las columnas indexadas no pueden tener valores NULL.

Cuando añades una columna AUTO_INCREMENT, los valores de columna se llenan con números secuenciales automáticamente. Puedes poner el primer valor de la secuencia ejecutando SET_INSERT_ID=# antes de ALTER TABLE o utilizando la opción de tabla AUTO_INCREMENT=#. Puedes ver la sección 5.5.6 [SET OPTION].

Con las tablas MyISAM, si no cambias la columna AUTO_INCREMENT, la secuencia de valores no se verá afectada. Si eliminas una columna AUTO_INCREMENT y luego añades otra AUTO_INCREMENT, los números empezarán desde 1 de nuevo.

Puedes ver la sección 4.6.1. [ALTER TABLE problems].

6.5.5. Sintaxis RENAME TABLE

RENAME TABLE table_name TO new_table_name [, table_name2 TO new_table_name2,…]

La renombración se hace automáticamente, lo que significa que ninguno otro ataque puede acceder a ninguna de las tablas mientras se está renombrando. Esto hace posible reemplazar la tabla por otra vacía:

CREATE TABLE new_table(...)
RENAME TABLE old_table TO backup_table, new_table TO old_table;

La renombración se hace de izquierda a derecha, lo que significa que si quieres intercambiar dos tablas, tienes que:

RENAME TABLE old_table TO backup_table,

```
        new_table TO old_table,
        backup_table TO new_table;
```

En la medida en la que dos bases de datos están en un mismo disco puedes también renombrar de una base de datos a otra:

```
RENAME TABLE current_db.table_name TO other_db.table_name;
```

Cuando ejecutas RENAME, no puedes tener tablas bloqueadas o transacciones en activo. Además debes tener privilegios de ALTER y DROP en la tabla original, y los privilegios de CREATE e INSERT en la nueva tabla.

Si MySQL encuentra errores en una renombración múltiple de tablas, realizará una renombración inversa para todas las tablas renombradas para volver atrás todo al estado original.

RENAME TABLE se añadión en MySQL 3.23.33.

6.5.6. Sintaxis DROP TABLE

```
DROP TABLE [IF EXISTS] table_name [, table_name,…][RESTRICT | CASCADE]
```

DROP TABLE elimina una o más tablas. Todos los datos de la tabla y su definición se eliminan, por lo que más vale que seas cuidadoso con este comando!

En MySQL versión 3.23 o posterior, puedes utilizar las palabras claves IF EXISTS para prevenir un error que ocurriría en tablas que no existieran.

RESTRICT y CASCADE se permite para una portabilidad más sencilla. Por el momento no hacen nada.

Nota: DROP TABLE soltará las transacciones actuales en activo.

6.5.7. Sintaxis CREATE INDEX

```
CREATE [UNIQUE | FULLTEXT] INDEX index_name
        ON table_name (col_name[(length)])
```

La sentencia CREATE INDEX no hace nada en las versiones anteriores a la 3.22. En la versión 3.22 o posterior, CREATE INDEX se dirige a una sentencia ALTER TABLE para crear índices. Puedes ver la sección 6.5.4. [ALTER TABLE].

Normalmente, puedes crear todos los índices en una tabla en el momento en el que la tabla es creada con CREATE TABLE. Puedes ver la sección 6.5.3 [CREATE TABLE]. CREATE INDEX te permite añadir índices a tablas existentes.

Una lista de columnas de la forma (col1,col2,…) crea un índice de múltiples columnas. Los valores de los índices se crean concatenando los valores de las columnas dadas.

Para las columnas CHAR y VARCHAR, los índices pueden crearse utilizando sólo una parte de la columna, utilizando la sintaxis col_name(length). (en las columnas BLOB o TEXT se requiere la longitud). La sentencia presentada aquí crea un índice utilizando los primeros 10 caracteres de la columna name:

```
mysql> CREATE INDEX part_of_name ON customer(name(10));
```

Dado que muchos nombres difieren en los primeros 10 caracteres, este índice no debería ser mucho más lento que el creado con la columna name entera. Además, utilizar columnas parciales para índices puede hacer que el archivo de índice tenga un tamaño mucho menor, lo que podría salvar un montón de disco, y acelerar además las operaciones de inserción!

Nota que puedes añadir un índice en una columna que puede tener valores NULL o en una columna BLOB/TEXT si estás utilizando la versión 3.23.2 de MySQL o posterior y utilizas el tipo de tabla MyISAM.

Para más información sobre cómo MySQL utiliza los índices, puedes ver la sección 5.4.3 [MySQL indexes].

Los índices FULLTEXT sólo pueden indexar columas VARCHAR o TEXT, y sólo en las tablas MyISAM. Los índices FULLTEXT están disponibles en la versión 3.23.23 y posterior. Puedes ver la sección 6.8. [Fulltext search].

6.5.8. Sintaxis DROP INDEX

```
DROP INDEX index_name ON table_name
```

DROP INDEX elimina el índice llamado index_name de la tabla table_name. DROP INDEX no hace nada en MySQL anterior a la version 3.22. En la versión 3.22 o posterior, DROP INDEX se dirige a una sentencia ALTER TABLE para eliminar el índice. Puedes ver la sección 6.5.4. [ALTER TABLE].

6.6. Comandos básicos MySQL de utilidad para el usuario

6.6.1. Sintaxis USE

```
USE db_name
```

La sentencia USE db_name le dice a MySQL que utilice la base de datos db_name como base de datos por defecto para las subsiguientes consultas. La base de datos se mantiene activa hasta el final de la sesión o hasta que se ejecute otra sentencia USE:

```
mysql> USE db1;
mysql> SELECT COUNT(*) FROM mytable;      # coge desde db1.mytable
mysql> USE db2;
mysql> SELECT COUNT(*) FROM mytable;      #coge desde db2.mytable
```

Haciendo una base de datos en concreto la activa mediante USE no te excluye de acceder a tablas de otra base de datos. El siguiente ejemploa accede a la tabla author desde la base de datos db1 y a la tabla editor de la base de datos db2.

```
mysql> USE db1;
mysql> SELECT author_name, editor_name FROM author, db2.editor
                WHERE author.editor_id=editor.editor_id;
```

La sentencia USE se provee por compatibilidad con Sybase

6.6.2. Sintaxis de DESCRIBE (Información sobre columnas)

{DESCRIBE |DESC} table_name {col_name | wild}

DESCRIBE es un acceso directo para SHOW COLUMNS FROM. Puedes ver la sección 4.5.6.1. [SHOW DATABASE INFO].

DESCRIBE provee información sobre las columnas de las tablas. col_name puede ser el nombre de una columna o una cadena que contiene los caracteres de comodín SQL '%' y '_'.

Si los tipos de columna son diferentes de lo que esperabas en base a la sentencia CREATE TABLE, nota que a veces MySQL cambia los tipos de columnas. Puedes ver la sección 6.5.3.1. [Silent column changes].

Esta sentencia se provee por compatibilidad con Oracle.
La sentencia SHOW aporta información similar. Puedes ver la sección 4.5.6.

6.7. Comandos transaccionales y de bloqueo de MySQL

6.7.1. Sintaxis de BEGIN/COMMIT/ROLLBACK

Por defecto MySQL funciona en modo autocommit. Esto implica que tan pronto como ejecutes una actualización, MySQL la almacenará en disco.

Si estás utilizando tablas a prueba de transacciones (como InnoDB, BDB), puedes poner MySQL en modo non-autocommit con el siguiente comando:

SET AUTOCOMMIT=0

Después de esto deberás hacer COMMIT para almacenar tus cambios en disco o ROLLBACK si quieres ignorar los cambios que has hecho desde el principio de la transacción.

Si quieres activar el modo AUTOCOMMIT para una serie de sentencias, puedes utilizar la sentencia BEGIN o BEGIN WORK:

```
BEGIN;
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
UPDATE table2 SET summary= @A WHERE type=1;
COMMIT;
```

Nota que si estás utilizando tablas que no son a prueba de transacciones, los cambios se almacenarán de uno en uno, independientemente del estado del modo autocommit.

Si haces un ROLLBACK cuando has actualizado una tabla no transaccional obtrendrás un error (ER_WARNING_NOT_COMPLETE_ROLLBACK) como alerta. Todas las tablas a prueba de transacciones volverán a como estaban, pero cualquier tabla no transaccional no cambiará.

Si estás utilizando BEGIN o SET AUTOCOMMIT=0, deberías utilizar log binario en vez del antiguo log de actualizaciones. Las transacciones son almacenadas en un log binario en un espacio, antes de COMMIT, para asegurarse que las transacciones echadas atrás no son almacenadas. Puedes ver la sección 4.9.4. [Binary log].

Los siguientes comandos automáticamente finalizan una transacción (Como si hubieras hecho un COMMIT antes de ejecutar el comando):

ALTER TABLE, BEGIN, CREATE INDEX, DROP DATABASE, DROP TABLE, RENAME TABLE, TRUNCATE

Puedes cambiar el nivel de aislamiento para las transacciones con SET TRANSACTION ISOLATION LEVEL. Puedes ver la sección 6.7.3. [SET TRANSACTION].

6.7.2. Sintaxis LOCK TABLES/UNLOCK TABLES

LOCK TABLES table_name [AS alias] {READ | [READ LOCAL] | [LOW PRIORITY] WRITE} [, table_name {READ | [READ LOCAL] | [LOW PRIORITY] WRITE…}
…
UNLOCK TABLES

LOCK TABLES bloquea tablas para la llamada actual. UNLOCK TABLE libera cualquier bloqueo mantenido por el acceso actual. Todas las tablas bloqueadas por la llamada actual son automáticamente desbloqueadas cuando el acceso realiza otro LOCK TABLES, o cuando la conexión con el servidor se cierra.

Para utilizar LOCK TABLES en MySQL 4.0.2. necesitas el privilegio global LOCK TABLES y el privilegio SELECT en las tablas implicadas. En MySQL 3.23. necesitas tener los privilegios SELECT, insert, DELETE y UPDATE para las tablas.

Las razones principales para utilizar LOCK TABLES ees emular transacciones o obtener más velocidad al actualizar las tablas. Esto se explica con más detalle posteriormente.

Si una llamada obtiene un LOCK de lectura en una tabla, esa llamada (y el resto de llamadas) sólo pueden leer los datos de la tabla. Si una llamada obtiene un bloqueo WRITE de una tabla, entonces la llamada que mantiene el bloqueo sólo puede realizar READ o WRITE sobre ésta. Las otras llamadas quedan bloqueadas.

La diferencia entre READ LOCAL y READ es que READ LOCAL permite la ejecución de sentencias INSERT no conflictivas mientras el bloqueo se mantiene. Sin enmbargo esto no puede ser utilizado si vas a manipular los archivos de la base de datos fuera de MySQL mientras mantienes el bloqueo.

Cuando utilizas LOCK TABLES, debes bloquear todas las tablas que vas a utilizar y debes usar el mismo alias que en las consultas! Si estás utilizando una tabla en varias ocasiones en una consulta (con alias), debes obtener un bloqueo por alias!

Los bloqueos WRITE normalmente tienen prioridad superior a los de tipo READ, para asegurar que las actualizaciones se procesan tan pronto como sea posible. Esto significa que si una llamada obtiene un bloqueo READ y otra llamada pide un bloqueo WRITE, los subsiguientes bloqueos READ esperarán hasta que la llamada WRITE haya obtenido el bloqueo y lo haya liberado. Puedes utilizar bloqueos LOW_PRIORITY WRITE para permitir que otras llamadas obtengan bloqueos READ mientras la llamada espera por el bloqueo WRITE. Sólo deberías usar bloqueos LOW_PRIORITY WRITE si estás seguro de que habrá eventualmente un momento en el que no habrá llamadas con un bloqueo READ.

LOCK TABLES funciona como sigue:

1. Ordena las tablas a bloquear en un orden definido internamente (desde el punto de vista del usuario, el orden es indefinido).
2. Si una tabla se bloquea con un bloqueo de lectura o escritura, pone el bloqueo de escritura antes del de lectura.
3. Se bloquea una tabla cada vez hasta que la llamada obtiene todos los bloqueos.

Esta política asegura que el bloqueo de la tabla está libre de puntos muertos. Hay otros aspectos por los que uno debe estar alerta en este esquema:

Si estás utilizando un bloqueo LOW_PRIORITY_WRITEpara una tabla, esto sólo significa sólo que MySQL esperará por este bloqueo en particular hasta que no haya bloqueos READ. Cuando la llamada ha obtenido el bloqueo WRITE y está esperando para obtener el bloqueo para la siguiente tabla en la lista de tablas a bloquear, las otras llamadas esperarán a que el bloqueo WRITE se libere. Si esto resulta en un serio problema con tu aplicación, deberías considerar de convertir algunas de tus tablas a tablas a prueba de transacciones.

Puedes matar una llamada que espera por un bloqueo de tabla con KILL. Puedes ver la sección 4.5.5. [KILL].

Nota que no deberías bloquear tablas para las que estás utilizando INSERT DELAYED. Esto se debe a que en este caso el INSERT se realiza por una llamada por separado.

Normalmente no tienes por qué bloquear tablas, ya que todas las sentencias sencillas UPDATE son atómicas; ninguna otra llamada puede interferir con cualquier otra sentencia SQL en ejecución. Hay algunos casos en los que podrías querer bloquear las tablas:

- Si vas a ejecutar varias operaciones en un montón de tablas, es mucho más rápido bloquear las tablas que vas a utilizar. La cruz es que, desde luego, ninguna otra llamada puede actualizar una tabla con bloqueo READ ni puede leer una tabla con bloqueo WRITE.

  La razón por la que algunas cosas funcionan más rápido bajo LOCK TABLES es que MySQL liberará el caché de claves para las tablas bloqueadas hasta que se llame a UNLOCK TABLES (normalmente el caché de claves se libera después de cada sentencia SQL). Esto acelera la inserción/actualización/borrado en tablas MyISAM.

- Si estás utilizando un manejador de tabla en MySQL que no soporta transacciones, debes utilizar LOCK TABLES si quieres asegurarte que no viene ninguna otra llamada entre un SELECT y un UPDATE. El ejemplo presentado aquí requiere LOCK TABLES para que se ejecute con seguridad:

  ```
  mysql> LOCK TABLES trans READ, customer WRITE;
  mysql> SELECT SUM(value) FROM trans WHERE customer_id=some_id;
  mysql> UPDATE customer SET total_value=sum_from_previous_statement
              WHERE customer_id=some_id;
  mysql> UNLOCK TABLES
  ```

  Sin UNLOCK TABLES, existe la posibilidad que otra llamada inserte un nuevo registro en la tabla trans entre la ejecución del SELECT y el UPDATE.

Utilizando actualizaciones incrementales (UPDATE customer SET value=value+new_value) o la función LAST_INSERT_ID(), puedes evitar el uso de LOCK TABLES en varios casos.

Puedes también resolver algunos casos utilizando las funciones de bloqueo a nivel de usuario GET_LOCK() y RELEASE_LOCK(). Estos bloqueos se guardan en una tabla hash en el servidor e implementada con pthread_mutex_lock() y pthread_mutex_unlock() para alta velocidad. Puedes ver la sección 6.3.6.2. [Miscellaneous functions].

Puedes ver la sección 5.3.1. [Internal locking] para más información sobre política de bloqueo.

Puedes bloquear todas las tablas de todas las bases de datos con bloqueos de lecturas con el comando FLUSH TABLES WITH READ LOCK. Puedes ver la sección 4.5.3. [FLUSH]. Esta es una forma muy conveniente para obtener copias de seguridad si tienes un sistema de archivos, como Veritas, que puede tomar snapshots en estos momentos.

NOTA: LOCK TABLES no está a prueba de transacciones y automáticamente va a ejecutar cualquier transacción activa antes de tratar de bloquear las tablas.

6.7.3. Sintaxis SET TRANSACTION

```
SET [GLOBAL|SESSION] TRANSACTION ISOLATION LEVEL
  {READ UNCOMMITTED | READ COMMITTED|REPEATABLE READ|SERIALIZABLE}
```

Sitúa el nivel de isolación de la transacción para el conjunto, sesión o siguiente transacción.

El comportamiento por defecto es dar el nivel de isolación para la siguiente transacción. Si utilizas la palabra clave GLOBAL, la sentencia pone el valor por defecto del nivel de isolación globalmente para todas las nuevas conexiones creadas desde ese momento. Necesitarás el privilegio SUPER para hacer esto. Utilizando la palabra clave SESSION se da el nivel de transacción por defecto para todas las futuras transacciones realizadas con la conexión actual.

Puedes dar el valor global de isolación para mysqld con --transaction-isolation=…Puedes ver la sección 4.1.1. [Command-line options].

6.8. Búsqueda de texto completo con MySQL

Desde la versión 3.23.23, MySQL tiene soporte para la búsqueda e indexación de texto completo. Los índices de texto completo en MySQL son índices del tipo FULLTEXT. Los índices FULLTEXT pueden ser creados desde columnas VARCHAR y TEXT en tiempo de CREATE TABLE o añadido posteriormente con ALTER TABLE o CREATE INDEX. Para conjuntos de datos amplios, será más rápido cargar tus datos en la tabla que no tiene índice FULLTEXT, entonces crear el índice con ALTER TABLE (o CREATE INDEX). Cargando los datos en la tabla con el índice ya creado será un proceso más lento.

La búsqueda de texto completo se realiza con la función MATCH().

```
mysql> CREATE TABLE articles (
        id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
        title VARCHAR(200),
        body TEXT
        FULLTEXT(title, body)
        );

Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO articles VALUES
```

```
(0,'MySQL Tutorial','DBMS stands for database...'),
(0,'How to use MySQL Efficiently','After you went through a...'),
(0,'Optimising MySQL','In this tutorial we will show...'),
(0,'1001 MySQL Trick','1. Never run mysqld as root 2. ...'),
(0,'MySQL vs. YourSQL','In the following database comparison...'),
(0,'MySQL security','When configured propuerly, MySQL...');
```

```
Query OK, 6 rows affected (0.00 sec)
Records: 6       Duplicates: 0              Warnings: 0
```

```
mysql> SELECT * FROM articles WHERE MATCH(title,body) AGAINST('database');
```

id	title	body
5	MySQL vs. YourSQL	In the following database comparison...
1	MySQL Tutorial	DBMS stands for DataBase...

2 rows in set (0.00 sec)

La función MATCH() realiza una búsqueda de lenguaje natural para una cadena contra una colección de texto (un conjunto de uno o más columnas incluídas en un índice FULLTEXT). La cadena de búsqueda se da como argumento a AGAINST(). La búsqueda se realiza en modo case-sensitive. Para cada registro de la tabla, MATCH() retorna un valor de relevancia, esto es, la medida de similitud entre la cadena de búsqueda y el texto en este registro de las columnas indicadas en la lista MATCH().

Cuando MATCH() se utiliza en una cláusula WHERE (puedes ver el ejemplo anterior) los registros retornados se ordenan automáticamente de mayor a menor relevancia. Los valores de relevancia son valores de coma flotante no negativos. Relevancia cero implica ninguna similitud. La relevancia se calcula basándose en el número de palabras del registro, el número de palabras únicas del registro, el número total de palabras de la colección, y el número de documentos (registros) que contienen una palabra particular.

También es posible realizar una búsqueda en modo booleano. Esto se explíca en la siguiente sección.

El ejemplo precedente es una presentación básica ilustrativa sobre cómo utilizar MATCH(). Los registros se retornan en orden de relevancia decreciente.

El siguiente ejemplo presenta como recuperar los valores de relevancia explícitamente. Como no están las cláusulas WHERE ni ORDER BY, los registros devueltos no están ordenados.

```
mysql> SELECT id, MATCH(title,blody) AGAINST('Tutorial') FROM articles;
```

id	MATCH(title,blody) AGAINST('Tutorial')
1	0.64840710366884
2	0
3	0.66266459031789
4	0
5	0
6	0

6 rows in set (0.00 sec)

El siguiente ejemplo es más complejo. La consulta retorna la relevancia y además ordena los registros en orden decreciente de relevancia. Para conseguir este resultado, deberías especificar MATCH() dos veces. Esto no causará un sobretrabajo adicional, porque el optimizador de MySQL indicara que las dos llamadas MATCH() son idénticas e invocan el código de la búsqueda a texto completo sólo por una vez.

```
mysql> SELECT id, body, MATCH(title,body) AGAINST
    ('Security implications of running MySQL as root') AS score
    FROM articles WHERE MATCH(title, body) AGAINST
    ('Security implications of running MySQL as root');
```

id	body	score
4	1. Never run mysqld as root. 2. …	1.5055546709332
6	When configured properly, MySQL	1.31140957288

2 rows in set (0.00 sec)

MySQL utiliza un intérprete muy simple para separar el texto en palabras. Una "palabra" es cualquier secuencia de caracteres consistente en letras, números, comillas simples y subrallados. Cualquier "palabra" presente en la lista de palabras de parada [stopword] o simplemente es muy corta (3 o menos caracteres) se ignora.

Cada palabra correcta de la colección y la consulta se pondera de acuerdo a su significación en la consulta o colección. De este modo, la palabra presente en varios documentos tendrá menor peso (e incluso puede tener peso cero), porque tiene menor valor semántico en esta colección particular. El cualquier otro caso, si la palabra es rara, recibirá un peso superior. Los pesos de las palabras son entonces combinados para computar la relevancia del registro.

Tal técnica funciona mejor con grandes colecciones (de hecho, se ajustó cuidadosamente de este modo). Para tablas muy pequeñas, la distribución de palabras no refleja adecuadamente su valor semántico, y este modelo puede producir ocasionalmente resultados extraños.

```
mysql> SELECT * FROM articles WHERE MATCH(title, body) AGAINS('MySQL');
Empty set (0.00 sec)
```

La búsqueda de la palabra MySQL no produce resultados en el ejemplo anterior, dado que la palabra está presente en más de la mitad de los registros. Como tal, se la trata como palabra de parada [stopword] (esto es, se trata de una palabra con valor semántico cero). Este es el comportamiento más deseable –una consulta de lenguaje natural no debería devolver cada segundo registro de una tabla de 1GB.

Una palabra que coincide en la mitad de registros es la menos verosímil para encontrar documentos relevantes. De hecho, es más probable que encuentre un montón de documentos irrelevantes. Todos sabemos que esto pasa demasiado a menudo cuando tratamos de encontrar algo en Internet con un motor de búsqueda. Es con este razonamiento que a tales registros se les ha asignado un valor semántico bajo en este **juego de datos particular**.

Desde la versión 4.0.1., MySQL puede realizar búsquedas booleanas de texto completo utilizando el modificador IN BOOLEAN MODE.

```
mysql> SELECT * FROM articles WHERE MATCH(title, body)
        AGAINST ('+MySQL - YourSQL' IN BOOLEAN MODE)
```

id	title	body
1	MySQL Tutorial	DBMS Stants for DataBase...
2	How To Use MySQL Efficiently	After you went through a…
3	Optimising MySQL	In this tutorial we will show…
4	1001 MySQL Tricks	1. Never run mysqld as root 2. …
6	MySQL Security	When configured properly, MySQL…

Esta consulta recuperó todos los registros que contienen la palabra MySQL (el umbral del 50% no se usa), pero que no contienen la palabra YourSQL. Nota que una búsqueda en modo booleano no ordena auto-mágicamente las filas en orden de relevancia decreciente. Puedes verlo en el resultado de la consulta precedente, donde la fila con la mayor relevancia (la que contiene MySQL dos veces) se halla en el último lugar, no en el primero. Una búsqueda booleana de texto completo también trabaja incluso sin el índice FULLTEXT, aunque eso sería lento.

La capacidad de la búsqueda booleana en texto completo soporta los siguientes operadores:

operador	significado
+	Al principio indica que esta palabra **debe** estar presente en los registros devueltos. Por defecto, la palabra es opcional, pero los registros que la contienen estarán mayor valoradas. Esto mimetiza el comportamiento de MATCH()…AGAINST() sin el modificador IN BOOLEAN MODE.
-	Al principio indica que la palabra **no debe** estar presente en los registros devueltos. Por defecto, la palabra es opcional, pero los registros que la contienen estarán mayor valoradas. Esto mimetiza el comportamiento de MATCH()…AGAINST() sin el modificador IN BOOLEAN MODE.
<>	Estos operadores se utilizan para cambiar la contribución de la palabra al valor de relevancia asignado a un registro. El operador < decrementa la contribución y el operador > lo incrementa. Puedes ver el ejemplo posterior.
()	Los paréntesis se utilizan para agrupar palabras en subexpresiones.
~	Al principio actúa como operador de negación, causando que la contribución de la palabra a la relevancia del resultado sea negativa. Es útil para marcar palabras ruidosas. El registro que contiene tal palabra se rateará menos que otras, pero no se exluirá, tal como pasaría con el operador -.
*	Se trata del operador de truncado. A diferencia de los otros operadores, debería estar añadido al final de la palabra, no al principio.
"	La frase que se encierra entre comillas dobles, encuentra registros concordantes sólo en el caso que la contengan **literalmente, como fue escrita**.

Y aquí hay algunos ejemplos:

texto	búsqueda
apple banana	Encuentra los registros que contienen al menos una de estas palabras.
+apple +juice	..ambas palabras
+apple macintosh	..palabra apple pero lo valorará más si contiene "macintosh"
+apple -macintosh	palabra "apple" pero no "macintosh"
+apple +(>pie <strudel)	…" apple" y "pie", o "apple" y "strudel" (en cualquier orden), pero valora "apple pie" más que "apple strudel"

apple*	…“ apple”, “apples”, “applesauce”, y “applet”
“ some words”	…“ some words of wisdom”, pero no “some noise words”

6.8.1. Restricciones del texto completo

- Todos los parámetros de la función MATCH() deben ser columnas de la misma tabla que es parte del mismo índice FULLTEXT, a menos que MATCH() sea IN BOOLEAN MODE.
- La lista de la columna MATCH() debe ajustarse exactamente a la lista de columnas de la definición del índice FULLTEXT para la tabla, a menos que MATCH() sea IN BOOLEAN MODE.
- El argumento de AGAINST() debe ser una cadena constante.

6.8.2. Calibrando la búsqueda de texto completo de MySQL

Desafortunadamente, la búsqueda a texto completo tiene pocos parámetros de usuario ajustables aún, a pesar de que añadir algunos es algo que se hará rápidamente. Si tienes una distribución de origen de MySQL (puedes ver la sección 2.3. [Installing source]), puedes ejercer mayor control sobre el comportamiento de la búsqueda a texto completo.

Nota que la búsqueda a texto completo fue ajustada cuidadosamente para la mejor efectividad buscadora. Modificando el comportamiento por defecto hará, en muchos casos, que los resultados de búsqueda empeoren. No alteres los datos de origen de MySQL si no sabes lo que estás haciendo!

- La longitud mínima de palabras a ser indexadas se define en la variable MySQL ft_min_word_len. Puedes ver la sección 4.5.6.4. [SHOW VARIABLES]. Cámbialo al valor que prefieras, y reconstruye tus índices FULLTEXT (esta variable sólo está disponible a partir de la versión 4.0 de MySQL).
- La lista de palabras de parada [stopword] se define en ‘myisam/ft_static.c’ modifícalo a tu gusto, recompila MySQL, y reconstruye tus índices FULLTEXT.
- El umbral del 50% se determina por el esquema de ponderación particular escogido. Para desactivarlo, cambia la siguiente línea en ‘myisam/ftdefs.h’:
  #define GWS_IN_USE GWS_PROB

  a:

  #define GWS_IN_USE GWS_FREQ

  Y recompila MySQL. No hay necesidad de reconstruir los índices en este caso. Nota: haciendo esto decrementas severamente la habilidad de MySQL para proveer valores de relevancia adecuados en la función MATCH(). Si realmente necesitas buscar por tales palabras comunes, sería mejor buscar utilizando IN BOOLEAN MODE, que no tiene en cuenta el umbral del 50%.

- Algunas veces el mantenedor del motor de búsqueda podría querer cambiar los operadores usados para las búsquedas a texto completo. Estos se definen en la variable ft_boolean_syntax. Puedes ver la sección 4.5.6.4. [SHOW VARIABLES]. Aún así, esta variable es de sólo lectura, y su valor se da en ‘myisam/ft_static.c’

6.8.3. Tareas en la búsqueda a texto completo

- Hacer todas las operaciones con el índice FULLTEXT más rápido.
- Operadores de proximidad.

- Soporte para “palabras de siempre índice”. Podrían ser cualquier cadena que usuario quiera tratar como palabras, tales como “C++”, “AS/400”, “TCP/IP”, etc.
- Soporte para búsqueda de texto completo en tablas MERGE.
- Soporte para juegos de caracteres multibyte.
- Hacer que la lista de palabras de parada dependa del lenguaje de los datos.
- Stemming (dependiendo del lenguaje de los datos, desde luego).
- Pre-intérprete de UDF genérico proveíble por el usuario.
- Flexibilizar el modelo (añadiendo parámetros ajustables a FULLTEXT en CREATE/ALTER TABLE).

6.9. Caché de consulta de MySQL

Desde la versión 4.0.1. MySQL server proporciona un caché de consulta. Al estar en uso, el caché de consulta almacena el texto de una consulta SELECT juntamente con el correspondiente resultado enviado al cliente. Si se recibe posteriormente una consulta idéntica, el servidor recuperará los resultados del caché antes que interpretar y ejecutar la misma consulta de nuevo.

NOTA: El caché de consulta no retorna datos alterados. Cuando los datos se modifican, cualquier entrada relevante en el caché de consulta se limpia.

El caché de consulta es extremamente útil en un entorno en el que (algunas) tablas no cambian muy a menudo y tienes un montón de consultas idénticas. Esta es una situación típica para varios servidores web que utilizan un montón de contenido dinámico.

Por debajo hay algunos datos de realización para el caché de consultas. (estos resultados fueron generados ejecutando el MySQL benchmark suit en Linux Alpha 2 x 5mm MHz con 2GB RAM y 64 MB caché de consulta):

- Si todas las consultas que estás realizando son simples (tales como seleccionar un registro de una tabla con un registro); pero se diferencian en que las consultas no pueden ser cacheadas, el sobretrabajo de tener el caché de consulta en activo es de 13%. Esto podría ser contemplado como el peor escenario. Sin embargo, en la vida real, las consultas son mucho más complicadas que nuestro simple ejemplo, por lo que el sobretrabajo es por regla general significativamente menor.
- Las búsquedas después de que un registro en una tabla con un solo registro es 238% más rápido. Esto puede ser contemplado como cercano al mínimo aumento de velocidad esperado para una consulta que se halla en caché.
- Si quieres desactivar el código de caché de consulta indica query_cache_size=0. Desactivando el caché de consulta no hay sobrecarga noticiable (el caché de consulta puede ser excluído del código con ayuda de la opción de configuración --without-query-cache).

6.9.1. Cómo opera el caché de consultas

Las consultas se comparan antes de la interpretación, así:

SELECT * FROM TABLE

y

Select * from table

son consideradas consultas diferentes para el caché de consultas, por lo que las consultas han de ser exactamente iguales (byte por byte) para ser vistas como idénticas. Además, una consulta puede ser considerada diferente si por ejemplo un cliente está usando un nuevo formato de protocolo de comunicación o cualquier juego de caracteres que otro cliente.

Las consultas que utilizan diferentes bases de datos, utilizan diferentes versiones de protocolos o que utiliza diferentes juegos de caracteres por defecto se consideran consultas diferentes y se guardan separadamente.

El caché funciona por los tipos de consulta SELECT CALC_ROWS… y SELECT FOUND_ROWS()…dado que el número de registros hallados se almacena también en el caché.

Si una tabla cambia (INSERT, UPDATE, DELETE, TRUNCATE, ALTER, o DROP TABLE| DATABASE), todas las consultas alojadas en el caché que utilizaron tal tabla (posiblemente a través de una tabla MRG_MyISAM!) son inválidas y se eliminan del caché.

Las tablas transaccionales InnoDB que han cambiado se invalidarán cuando se realice un COMMIT.

Una consulta tampoco puede ser alojada en caché si contiene una de estas funciones:

Función	Función	Función
Definida por usuario	CONNECTION_ID	FOUND_ROWS
GET_LOCK	RELEASE_LOCK	LOAD_FILE
MASTER_POS_WAIT	NOW	SYSDATE
CURRENT_TIMESTAMP	CURDATE	CURRENT_DATE
CURTIME	CURRENT_TIME	DATABASE
ENCRYPT(con parámetro)	LAST_INSERT_ID	RAND
UNIX_TIMESTAMP (sin parámetros)	USER	BENCHMARK

Tampoco puede ser guardada una consulta si contiene variables de usuario, si es de la forma SELECT…IN SHARE MODE o de la forma SELECT * FROM AUTOINCREMENT_FIELD IS NULL (para recuperar el último id de inserción en el entorno de trabajo de ODBC).

Sin embargo, FOUND ROWS() retornará el valor correcto, inculso si la consulta precedente fue recuperada del caché.

Las consultas que no utilizan tablas o en las que el usuario tiene un privilegio de columna para cualquier a de loas tablas implicadas, tampoco se cachean.

Antes de que una consulta es recogida del caché de consulta, MySQL chequeará que el usuario tiene el privilegio SELECT para todas las bases de datos y tablas implicadas. Si no es el caso, el resultado cacheado no se utilizará.

6.9.2. Configuración del caché de consulta

El caché de consulta de MySQL añade una nueva variable de sistema para mysqld que puede ser dada en el archivo de configuración, el la línea de comandos al iniciar mysqld.

- query_cache_limit no cachea los resultados mayores de este (por defecto 1MB).
- query_cache_size. La memoria alojada para almacenar los resultados de antiguas consultas. Si es 0, el caché de consulta está desactivado (valor por defecto).

- query_cache_type. Se le puede dar los valores siguientes (sólo numéricos):

  0: OFF, no cachea o recupera resultados.
  1: ON cachea todos los resultados excepto consultas SELECT SQL_NO_CACHE.
  2: DEMAND, consultas de sólo caché SELECT SQL_CACHE…

Dentro de una llamada (conexión), el comportamiento del caché de consulta puede cambiar sobre el existente por defecto. La sintaxis es como sigue:

QUERY_CACHE_TYPE=OFF|ON|DEMAND QUERY_CACHE_TYPE=0|1|2

0 o OFF: No cachea los resultados recuperados.
1 o ON: Cachea todos los resultados excepto consultas SELECT SQL_NO_CACHE…
2 o DEMAND: Sólo cachea resultados en consultas SELECT SQL_CACHE…

6.9.3. Opciones del caché de consulta en SELECT

Hay dos parámetros posibles para el cache de consulta que pueden especificarse en una consulta SELECT:

Opción	Descripción
SQL_CACHE	Si QUERY_CACHE_TYPE es DEMAND, Permite a la consulta ser cacheada. Si QUERY_CACHE_TYPE está en ON, esto es el valor por defecto. Si QUERY_CACHE_TYPE es OFF, no hace nada.
SQL_NO_CACHE	Hace que la consulta no sea cacheable, no permite a la consulta que sea almacenada en el caché.

6.9.4. Estado y mantenimiento del caché de consultas

Con el comando FLUSH QUERY CACHE puedes defragmentar el caché de consultas para utilizar mejor su memoria. Este comando no borrará las consultas del caché. FLUSH TABLES libera el caché de consulta.

El comando RESET QUERY CACHE borra todas los resultados de consultas del caché. Puedes monitorizar la realización del caché de consulta en SHOW STATUS:

Variable	Descripción
Qcache_queries_in_cache	Número de consultas registradas en el caché
Qcache_inserts	Número de consultas añadidas al caché
Qcache_hits	Número de hits del caché
Qcache_no_cached	Número de consultas no cacheadas (no cacheables o debido a QUERY_CACHE_TYPE)
Qcache_free_memory	Volumen de memoria libre para caché de consulta
Qcache_total_blocks	Número total de bloques del caché de consulta
Qcache_free_blocks	Número de bloques libres del caché de consulta

Número total de consultas = Qcache_inserts + Qcache_hits+Qcache_not_cached

El caché de consultas utiliza bloques de longitud variable, por lo que las variables Qcache_total_blocks y Qcache_free_blocks pueden indicar la fragmentación de la memoria del cache de consulta. Después de FLUSH QUERY CACHE sólo se devuelven los bloques individuales (libres).

Nota: Cada consulta necesita un mínimo de dos bloques (uno para el texto de la consulta y uno o mas para el resultado de la consulta). Además, cada tabla utilizada por la consulta necesita un bloque, pero si dos o más consultas utilizan la misma tabla, sólo se necesita un bloque para alojarla.