

# MySQL language reference

## Literals: how to write strings and numbers

### Strings

A string is a sequence of characters, surrounded by either single quote (``'``) or double quote (``"``) characters. Examples:

```
'a string'  
"another string"
```

Within a string, certain sequences have special meaning. Each of these sequences begins with a backslash (``\``), known as the *escape character*. **MySQL** recognizes the following escape sequences:

```
\0    An ASCII 0 (NUL) character.  
\n    A newline character.  
\t    A tab character.  
\r    A carriage return character.  
\b    A backspace character.  
\'    A single quote (``'``) character.  
\''   A double quote (``"``) character.  
\\\   A backslash (``\``) character.  
\%    A ``%`` character. This is used to search for literal instances of ``%`` in contexts where ``%`` would otherwise be interpreted as a wildcard character.  
\_    A ``_`` character. This is used to search for literal instances of ``_`` in contexts where ``_`` would otherwise be interpreted as a wildcard character.
```

There are several ways to include quotes within a string:

- A ``'`` inside a string quoted with ``'`` may be written as ``\'``.
- A ``"`` inside a string quoted with ``"`` may be written as ``\"``.
- You can precede the quote character with an escape character (``\``).
- A ``'`` inside a string quoted with ``"`` needs no special treatment and need not be doubled or escaped. In the same way, ``"`` inside a string quoted with ``'`` needs no special treatment.

The `SELECT` statements shown below demonstrate how quoting and escaping work:

```
mysql> SELECT 'hello', '"hello"', '"hello"', 'hel"lo', '\hello';  
+-----+-----+-----+-----+-----+  
| hello | "hello" | "hello" | hel"lo | 'hello |  
+-----+-----+-----+-----+-----+  
  
mysql> SELECT "hello", "'hello'", "hello", "hel"lo", "\"hello";  
+-----+-----+-----+-----+-----+  
| hello | 'hello' | hello  | hel"lo | "hello |  
+-----+-----+-----+-----+-----+  
  
mysql> SELECT "This\nIs\nFour\nlines";  
+-----+  
| This  
+-----+
```

```
Is
Four
lines |
+-----+
```

If you want to insert binary data into a BLOB column, the following characters must be represented by escape sequences:

NUL ASCII 0. You should represent this by ``\0` (a backslash and an ASCII ``0'` character).  
\ `ASCII 92, backslash. Represent this by \\.  
' ASCII 39, single quote. Represent this by \'.  
" ASCII 34, double quote. Represent this by \".`

If you write C code, you can use the C API function `mysql_escape_string()` to escape characters for the `INSERT` statement. See section [19.3 C API function overview](#). In Perl, you can use the `quote` method of the DBI package to convert special characters to the proper escape sequences. See section [19.5.2 The DBI interface](#).

You should use an escape function on any string that might contain any of the special characters listed above!

## Numbers

Integers are represented as a sequence of digits. Floats use ``.`` as a decimal separator. Either type of number may be preceded by `-` to indicate a negative value.

Examples of valid integers:

```
1221
0
-32
```

Examples of valid floating-point numbers:

```
294.42
-32032.6809e+10
148.00
```

An integer may be used in a floating-point context; it is interpreted as the equivalent floating-point number.

## NULL values

The `NULL` value means "no data" and is different from values such as 0 for numeric types or the empty string for string types. See section [17.14 Problems with NULL values](#).

`NULL` may be represented by `\N` when using the text file import or export formats (`LOAD DATA INFILE`, `SELECT ... INTO OUTFILE`). See section [LOAD DATA INFILE syntax](#).

## Database, table, index, column and alias names

Database, table, index, column and alias names all follow the same rules in **MySQL**:

- A name may consist of alphanumeric characters from the current character set and also `'\_` and `'\$`. The default character set is ISO-8859-1 Latin1; this may be changed by recompiling **MySQL**. See section 9.1.1 [The character set used for data and sorting](#).
- A database, table, index or column name can be up to 64 characters long. An alias name can be up to 256 characters long.
- A name may start with any character that is legal in a name. In particular, a name may start with a number (this differs from many other database systems!). However, a name cannot consist *only* of numbers.
- It is recommended that you do not use names like `1e`, because an expression like `1e+1` is ambiguous. It may be interpreted as the expression `1e + 1` or as the number `1e+1`.
- You cannot use the `.` character in names because it is used to extend the format by which you can refer to columns (see immediately below).

In **MySQL** you can refer to a column using any of the following forms:

Column reference	Meaning
<code>col_name</code>	Column <code>col_name</code> from whichever table used in the query contains a column of that name
<code>tbl_name.col_name</code>	Column <code>col_name</code> from table <code>tbl_name</code> of the current database
<code>db_name.tbl_name.col_name</code>	Column <code>col_name</code> from table <code>tbl_name</code> of the database <code>db_name</code> . This form is available in <b>MySQL</b> 3.22 or later.

You need not specify a `tbl_name` or `db_name.tbl_name` prefix for a column reference in a statement unless the reference would be ambiguous. For example, suppose tables `t1` and `t2` each contain a column `c`, and you retrieve `c` in a `SELECT` statement that uses both `t1` and `t2`. In this case, `c` is ambiguous because it is not unique among the tables used in the statement, so you must indicate which table you mean by writing `t1.c` or `t2.c`. Similarly, if you are retrieving from a table `t` in database `db1` and from a table `t` in database `db2`, you must refer to columns in those tables as `db1.t.col_name` and `db2.t.col_name`.

The syntax `.tbl_name` means the table `tbl_name` in the current database. This syntax is accepted for ODBC compatibility, because some ODBC programs prefix table names with a `.` character.

### Case sensitivity in names

In **MySQL**, databases and tables correspond to directories and files within those directories. Consequently, the case sensitivity of the underlying operating system determines the case sensitivity of database and table names. This means database and table names are case sensitive in Unix and case insensitive in Win32.

**Note:** Although database and table names are case insensitive for Win32, you should not refer to a given database or table using different cases within the same query. The following query would not work because it refers to a table both as `my_table` and as `MY_TABLE`:

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

Column names are case insensitive in all cases.

Aliases on tables are case sensitive. The following query would not work because it refers to the alias both as `a` and as `A`:

```
mysql> SELECT col_name FROM tbl_name AS a
        WHERE a.col_name = 1 OR A.col_name = 2;
```

Aliases on columns are case insensitive.

## Column types

**MySQL** supports a number of column types, which may be grouped into three categories: numeric types, date and time types, and string (character) types. This section first gives an overview of the types available and summarizes the storage requirements for each column type, then provides a more detailed description of the properties of the types in each category. The overview is intentionally brief. The more detailed descriptions should be consulted for additional information about particular column types, such as the allowable formats in which you can specify values.

The column types supported by **MySQL** are listed below. The following code letters are used in the descriptions:

- M Indicates the maximum display size. The maximum legal display size is 255.
- D Applies to floating-point types and indicates the number of digits following the decimal point.

Square brackets (``[ ' and ` ] '`) indicate parts of type specifiers that are optional.

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

A very small integer. The signed range is -128 to 127. The unsigned range is 0 to 255.

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

A small integer. The signed range is -32768 to 32767. The unsigned range is 0 to 65535.

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

A medium-size integer. The signed range is -8388608 to 8388607. The unsigned range is 0 to 16777215.

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

A normal-size integer. The signed range is -2147483648 to 2147483647. The unsigned range is 0 to 4294967295.

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

This is a synonym for `INT`.

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

A large integer. The signed range is -9223372036854775808 to 9223372036854775807. The unsigned range is 0 to 18446744073709551615. Note that all arithmetic is done using signed `BIGINT` or `DOUBLE` values, so you shouldn't use unsigned big integers larger than 9223372036854775807 (63 bits) except with bit functions! Note that `-`, `+` and `*` will use `BIGINT` arithmetic when both arguments are `INTEGER` values! This means that if you multiply two big integers (or results from functions that return integers) you may get unexpected results if the result is larger than 9223372036854775807.

`FLOAT(precision) [ZEROFILL]`

A floating-point number. Cannot be unsigned. `precision` can be 4 or 8. `FLOAT(4)` is a single-precision number and `FLOAT(8)` is a double-precision number. These types are like the `FLOAT` and `DOUBLE` types described immediately below. `FLOAT(4)` and `FLOAT(8)` have the same

ranges as the corresponding `FLOAT` and `DOUBLE` types, but their display size and number of decimals is undefined. In **MySQL 3.23**, this is a true floating point value. In earlier **MySQL** versions, `FLOAT(precision)` always has 2 decimals. This syntax is provided for ODBC compatibility.

`FLOAT(M,D) [ZEROFILL]`

A small (single-precision) floating-point number. Cannot be unsigned. Allowable values are `-3.402823466E+38` to `-1.175494351E-38`, 0 and `1.175494351E-38` to `3.402823466E+38`.

`DOUBLE(M,D) [ZEROFILL]`

A normal-size (double-precision) floating-point number. Cannot be unsigned. Allowable values are `-1.7976931348623157E+308` to `-2.2250738585072014E-308`, 0 and `2.2250738585072014E-308` to `1.7976931348623157E+308`.

`DOUBLE PRECISION(M,D) [ZEROFILL]`

`REAL(M,D) [ZEROFILL]`

These are synonyms for `DOUBLE`.

`DECIMAL(M,D) [ZEROFILL]`

An unpacked floating-point number. Cannot be unsigned. Behaves like a `CHAR` column: "unpacked" means the number is stored as a string, using one character for each digit of the value, the decimal point, and, for negative numbers, the '-' sign. If `D` is 0, values will have no decimal point or fractional part. The maximum range of `DECIMAL` values is the same as for `DOUBLE`, but the actual range for a given `DECIMAL` column may be constrained by the choice of `M` and `D`. In **MySQL 3.23** the `M` argument no longer includes the sign or the decimal point. (This is according to ANSI SQL.)

`NUMERIC(M,D) [ZEROFILL]`

This is a synonym for `DECIMAL`.

`DATE` A date. The supported range is '1000-01-01' to '9999-12-31'. **MySQL** displays `DATE` values in 'YYYY-MM-DD' format, but allows you to assign values to `DATE` columns using either strings or numbers.

`DATETIME`

A date and time combination. The supported range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. **MySQL** displays `DATETIME` values in 'YYYY-MM-DD HH:MM:SS' format, but allows you to assign values to `DATETIME` columns using either strings or numbers.

`TIMESTAMP(M)`

A timestamp. The range is '1970-01-01 00:00:00' to sometime in the year 2037. **MySQL** displays `TIMESTAMP` values in `YYYYMMDDHHMMSS`, `YYMMDDHHMMSS`, `YYYYMMDD` or `YYMMDD` format, depending on whether `M` is 14 (or missing), 12, 8 or 6, but allows you to assign values to `TIMESTAMP` columns using either strings or numbers. A `TIMESTAMP` column is useful for recording the date and time of an `INSERT` or `UPDATE` operation because it is automatically set to the date and time of the most recent operation if you don't give it a value yourself. You can also set it to the current date and time by assigning it a `NULL` value. See section [Date and time types](#).

`TIME` A time. The range is '-838:59:59' to '838:59:59'. **MySQL** displays `TIME` values in 'HH:MM:SS' format, but allows you to assign values to `TIME` columns using either strings or numbers.

`YEAR` A year. The allowable values are 1901 to 2155, and 0000. **MySQL** displays `YEAR` values in `YYYY` format, but allows you to assign values to `YEAR` columns using either strings or numbers. (The `YEAR` type is new in **MySQL 3.22**.)

`CHAR(M) [BINARY]`

A fixed-length string that is always right-padded with spaces to the specified length when stored. The range of `M` is 1 to 255 characters. Trailing spaces are removed when the value is retrieved. `CHAR` values are sorted and compared in case-insensitive fashion unless the `BINARY` keyword is

given.

`VARCHAR(M) [BINARY]`

A variable-length string. Note: Trailing spaces are removed when the value is stored (this differs from the ANSI SQL specification). The range of `M` is 1 to 255 characters. `VARCHAR` values are sorted and compared in case-insensitive fashion unless the `BINARY` keyword is given. See section [Silent column specification changes](#).

`TINYBLOB`

`TINYTEXT`

A `BLOB` or `TEXT` column with a maximum length of 255 ( $2^8 - 1$ ) characters. See section [Silent column specification changes](#).

`BLOB`

`TEXT` A `BLOB` or `TEXT` column with a maximum length of 65535 ( $2^{16} - 1$ ) characters. See section [Silent column specification changes](#).

`MEDIUMBLOB`

`MEDIUMTEXT`

A `BLOB` or `TEXT` column with a maximum length of 16777215 ( $2^{24} - 1$ ) characters. See section [Silent column specification changes](#).

`LOBLOB`

`LONGTEXT`

A `BLOB` or `TEXT` column with a maximum length of 4294967295 ( $2^{32} - 1$ ) characters. See section [Silent column specification changes](#).

`ENUM('value1', 'value2', ...)`

An enumeration. A string object that can have only one value, chosen from the list of values 'value1', 'value2', ..., or `NULL`. An `ENUM` can have a maximum of 65535 distinct values.

`SET('value1', 'value2', ...)`

A set. A string object that can have zero or more values, each of which must be chosen from the list of values 'value1', 'value2', ... A `SET` can have a maximum of 64 members.

## Column type storage requirements

The storage requirements for each of the column types supported by **MySQL** are listed below by category.

### 7.2.2 Numeric types

Column type	Storage required
TINYINT	1 byte
SMALLINT	2 bytes
MEDIUMINT	3 bytes
INT	4 bytes
INTEGER	4 bytes
BIGINT	8 bytes
FLOAT(4)	4 bytes
FLOAT(8)	8 bytes
FLOAT	4 bytes
DOUBLE	8 bytes
DOUBLE PRECISION	8 bytes
REAL	8 bytes
DECIMAL(M,D)	M bytes (D+2, if M < D)
NUMERIC(M,D)	M bytes (D+2, if M < D)

### 7.2.3 Date and time types

Column type	Storage required
DATETIME	8 bytes
DATE	3 bytes
TIMESTAMP	4 bytes
TIME	3 bytes
YEAR	1 byte

### 7.2.4 String types

Column type	Storage required
CHAR(M)	M bytes, 1 ≤ M ≤ 255
VARCHAR(M)	L+1 bytes, where L ≤ M and 1 ≤ M ≤ 255
TINYBLOB, TINYTEXT	L+1 bytes, where L < 2 <sup>8</sup>
BLOB, TEXT	L+2 bytes, where L < 2 <sup>16</sup>
MEDIUMBLOB, MEDIUMTEXT	L+3 bytes, where L < 2 <sup>24</sup>
LOB, LONGTEXT	L+4 bytes, where L < 2 <sup>32</sup>
ENUM('value1', 'value2', ...)	1 or 2 bytes, depending on the number of enumeration values (65535 values maximum)
SET('value1', 'value2', ...)	1, 2, 3, 4 or 8 bytes, depending on the number of set members (64 members maximum)

`VARCHAR` and the `BLOB` and `TEXT` types are variable-length types, for which the storage requirements depend on the actual length of column values (represented by `L` in the preceding table), rather than on the type's maximum possible size. For example, a `VARCHAR(10)` column can hold a string with a maximum length of 10 characters. The actual storage required is the length of the string (`L`), plus 1 byte to record the length of the string. For the string `'abcd'`, `L` is 4 and the storage requirement is 5 bytes.

The `BLOB` and `TEXT` types require 1, 2, 3 or 4 bytes to record the length of the column value, depending on the maximum possible length of the type.

If a table includes any variable-length column types, the record format will also be variable-length. Note that when a table is created, **MySQL** may under certain conditions change a column from a variable-length type to a fixed-length type, or vice-versa. See section [Silent column specification changes](#).

The size of an `ENUM` object is determined by the number of different enumeration values. 1 byte is used for enumerations with up to 255 possible values. 2 bytes are used for enumerations with up to 65535 values.

The size of a `SET` object is determined by the number of different set members. If the set size is `N`, the object occupies  $(N+7)/8$  bytes, rounded up to 1, 2, 3, 4 or 8 bytes. A `SET` can have a maximum of 64 members.

## Numeric types

All integer types can have an optional attribute `UNSIGNED`. Unsigned values can be used when you want to allow only positive numbers in a column and you need a little bigger numeric range for the column.

All numeric types can have an optional attribute `ZEROFILL`. Values for `ZEROFILL` columns are left-padded with zeroes up to the maximum display length when they are displayed. For example, for a column declared as `INT(5) ZEROFILL`, a value of 4 is retrieved as 00004.

When asked to store a value in a numeric column that is outside the column type's allowable range, **MySQL** clips the value to the appropriate endpoint of the range and stores the resulting value instead.

For example, the range of an `INT` column is -2147483648 to 2147483647. If you try to insert -9999999999 into an `INT` column, the value is clipped to the lower endpoint of the range, and -2147483648 is stored instead. Similarly, if you try to insert 9999999999, 2147483647 is stored instead.

If the `INT` column is `UNSIGNED`, the size of the column's range is the same but its endpoints shift up to 0 and 4294967295. If you try to store -9999999999 and 9999999999, the values stored in the column become 0 and 4294967296.

Conversions that occur due to clipping are reported as "warnings" for `ALTER TABLE`, `LOAD DATA INFILE`, `UPDATE` and multi-row `INSERT` statements.

The maximum display size (`M`) and number of decimals (`D`) are used for formatting and calculation of maximum column width.

**MySQL** will store any value that fits a column's storage type even if the value exceeds the display size.

For example, an `INT(4)` column has a display size of 4. Suppose you insert a value which has more than 4 digits into the column, such as 12345. The display size is exceeded, but the allowable range of the `INT` type is not, so **MySQL** stores the actual value, 12345. When retrieving the value from the column, **MySQL** returns the actual value stored in the column.

The `DECIMAL` type is considered a numeric type (as is its synonym, `NUMERIC`), but such values are stored as strings. One character is used for each digit of the value, the decimal point (if `D > 0`) and the '-' sign (for negative numbers). If `D` is 0, `DECIMAL` and `NUMERIC` values contain no decimal point or fractional part.

The maximum range of `DECIMAL` values is the same as for `DOUBLE`, but the actual range for a given `DECIMAL` column may be constrained by the choice of `M` and `D`. For example, a type specification such as `DECIMAL(4,2)` indicates a maximum length of four characters with two digits after the decimal point. Due to the way the `DECIMAL` type is stored, this specification results in an allowable range of `- .99` to `9 .99`, much less than the range of a `DOUBLE`.

To avoid some rounding problems, **MySQL** always rounds everything that it stores in any floating-point column to the number of decimals indicated by the column specification. Suppose you have a column type of `FLOAT(8,2)`. The number of decimals is 2, so a value such as `2 .333` is rounded to two decimals and stored as `2 .33`.

## Date and time types

The date and time types are `DATETIME`, `DATE`, `TIMESTAMP`, `TIME` and `YEAR`. Each of these has a range of legal values, as well as a "zero" value that is used when you specify an illegal value.

Here are some general considerations to keep in mind when working with date and time types:

- **MySQL** retrieves values for a given date or time type in a standard format, but it attempts to interpret a variety of formats for values that you supply (e.g., when you specify a value to be assigned to or compared to a date or time type). Nevertheless, only the formats described in the following sections are supported. It is expected that you will supply legal values, and unpredictable results may occur if you use values in other formats.
- Although **MySQL** tries to interpret values in several formats, it always expects the year part of date values to be leftmost. Dates must be given in year-month-day order (e.g., `'98-09-04'`), rather than in the month-day-year or day-month-year orders commonly used elsewhere (e.g., `'09-04-98'`, `'04-09-98'`).
- **MySQL** automatically converts a date or time type value to a number if the value is used in a numeric context, and vice versa.
- When **MySQL** encounters a value for a date or time type that is out of range or otherwise illegal for the type, it converts the value to the "zero" value for that type. (The exception is that out-of-range `TIME` values are clipped to the appropriate endpoint of the `TIME` range.) The table below shows the format of the "zero" value for each type:

Column type	"Zero" value
DATETIME	'0000-00-00 00:00:00'
DATE	'0000-00-00'
TIMESTAMP	00000000000000 (length depends on display size)
TIME	'00:00:00'
YEAR	0000

- The "zero" values are special, but you can store or refer to them explicitly using the values shown in the table. You can also do this using the values '0' or 0, which are easier to write.
- "Zero" date or time values used through **MyODBC** are converted automatically to NULL in **MyODBC 2.50.12** and above, because ODBC can't handle such values.

## Y2K issues and date types

**MySQL** itself is Y2K-safe (see section [1.6 Year 2000 compliance](#)), but input values presented to **MySQL** may not be. Any input containing 2-digit year values is ambiguous, since the century is unknown. Such values must be interpreted into 4-digit form since **MySQL** stores years internally using four digits.

For DATETIME, DATE, TIMESTAMP and YEAR types, **MySQL** interprets dates with ambiguous year values using the following rules:

- Year values in the range 00-69 are converted to 2000-2069.
- Year values in the range 70-99 are converted to 1970-1999.

Remember that these rules provide only reasonable guesses as to what your data mean. If the heuristics used by **MySQL** don't produce the correct values, you should provide unambiguous input containing 4-digit year values.

## The DATETIME, DATE and TIMESTAMP types

The DATETIME, DATE and TIMESTAMP types are related. This section describes their characteristics, how they are similar and how they differ.

The DATETIME type is used when you need values that contain both date and time information. **MySQL** retrieves and displays DATETIME values in 'YYYY-MM-DD HH:MM:SS' format. The supported range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. ("Supported" means that although earlier values might work, there is no guarantee that they will.)

The DATE type is used when you need only a date value, without a time part. **MySQL** retrieves and displays DATE values in 'YYYY-MM-DD' format. The supported range is '1000-01-01' to '9999-12-31'.

The TIMESTAMP column type provides a type that you can use to automatically mark INSERT or UPDATE operations with the current date and time. If you have multiple TIMESTAMP columns, only the first one is updated automatically.

Automatic updating of the first TIMESTAMP column occurs under any of the following conditions:

- The column is not specified explicitly in an `INSERT` or `LOAD DATA INFILE` statement.
- The column is not specified explicitly in an `UPDATE` statement and some other column changes value. (Note that an `UPDATE` that sets a column to the value it already has will not cause the `TIMESTAMP` column to be updated, because if you set a column to its current value, **MySQL** ignores the update for efficiency.)
- You explicitly set the `TIMESTAMP` column to `NULL`.

`TIMESTAMP` columns other than the first may also be set to the current date and time. Just set the column to `NULL`, or to `NOW()`.

You can set any `TIMESTAMP` column to a value different than the current date and time by setting it explicitly to the desired value. This is true even for the first `TIMESTAMP` column. You can use this property if, for example, you want a `TIMESTAMP` to be set to the current date and time when you create a row, but not to be changed whenever the row is updated later:

- Let **MySQL** set the column when the row is created. This will initialize it to the current date and time.
- When you perform subsequent updates to other columns in the row, set the `TIMESTAMP` column explicitly to its current value.

On the other hand, you may find it just as easy to use a `DATETIME` column that you initialize to `NOW()` when the row is created and leave alone for subsequent updates.

`TIMESTAMP` values may range from the beginning of 1970 to sometime in the year 2037, with a resolution of one second. Values are displayed as numbers.

The format in which **MySQL** retrieves and displays `TIMESTAMP` values depends on the display size, as illustrated by the table below. The 'full' `TIMESTAMP` format is 14 digits, but `TIMESTAMP` columns may be created with shorter display sizes:

Column type	Display format
<code>TIMESTAMP ( 14 )</code>	YYYYMMDDHHMMSS
<code>TIMESTAMP ( 12 )</code>	YYMMDDHHMMSS
<code>TIMESTAMP ( 10 )</code>	YYMMDDHHMM
<code>TIMESTAMP ( 8 )</code>	YYYYMMDD
<code>TIMESTAMP ( 6 )</code>	YYMMDD
<code>TIMESTAMP ( 4 )</code>	YYMM
<code>TIMESTAMP ( 2 )</code>	YY

All `TIMESTAMP` columns have the same storage size, regardless of display size. The most common display sizes are 6, 8, 12, and 14. You can specify an arbitrary display size at table creation time, but values of 0 or greater than 14 are coerced to 14. Odd-valued sizes in the range from 1 to 13 are coerced to the next higher even number.

You can specify `DATETIME`, `DATE` and `TIMESTAMP` values using any of a common set of formats:

- As a string in either 'YYYY-MM-DD HH:MM:SS' or 'YY-MM-DD HH:MM:SS' format. A "relaxed"

syntax is allowed--any non-numeric character may be used as the delimiter between date parts or time parts. For example, '98-12-31 11:30:45', '98.12.31 11+30+45', '98/12/31 11\*30\*45' and '98@12@31 11^30^45' are equivalent.

- As a string in either 'YYYY-MM-DD' or 'YY-MM-DD' format. A "relaxed" syntax is allowed here, too. For example, '98-12-31', '98.12.31', '98/12/31' and '98@12@31' are equivalent.
- As a string with no delimiters in either 'YYYYMMDDHHMMSS' or 'YYMMDDHHMMSS' format, provided that the string makes sense as a date. For example, '19970523091528' and '970523091528' are interpreted as '1997-05-23 09:15:28', but '971122459015' is illegal (it has a nonsensical minute part) and becomes '0000-00-00 00:00:00'.
- As a string with no delimiters in either 'YYYYMMDD' or 'YYMMDD' format, provided that the string makes sense as a date. For example, '19970523' and '970523' are interpreted as '1997-05-23', but '971332' is illegal (it has nonsensical month and day parts) and becomes '0000-00-00'.
- As a number in either YYYYMMDDHHMMSS or YYMMDDHHMMSS format, provided that the number makes sense as a date. For example, 19830905132800 and 830905132800 are interpreted as '1983-09-05 13:28:00'.
- As a number in either YYYYMMDD or YYMMDD format, provided that the number makes sense as a date. For example, 19830905 and 830905 are interpreted as '1983-09-05'.
- As the result of a function that returns a value that is acceptable in a DATETIME, DATE or TIMESTAMP context, such as NOW() or CURRENT\_DATE.

Illegal DATETIME, DATE or TIMESTAMP values are converted to the "zero" value of the appropriate type ('0000-00-00 00:00:00', '0000-00-00' or 000000000000000).

For values specified as strings that include date part delimiters, it is not necessary to specify two digits for month or day values that are less than 10. '1979-6-9' is the same as '1979-06-09'. Similarly, for values specified as strings that include time part delimiters, it is not necessary to specify two digits for hour, month or second values that are less than 10. '1979-10-30 1:2:3' is the same as '1979-10-30 01:02:03'.

Values specified as numbers should be 6, 8, 12 or 14 digits long. If the number is 8 or 14 digits long, it is assumed to be in YYYYMMDD or YYYYMMDDHHMMSS format and that the year is given by the first 4 digits. If the number is 6 or 12 digits long, it is assumed to be in YYMMDD or YYMMDDHHMMSS format and that the year is given by the first 2 digits. Numbers that are not one of these lengths are interpreted as though padded with leading zeros to the closest length.

Values specified as non-delimited strings are interpreted using their length as given. If the string is 8 or 14 characters long, the year is assumed to be given by the first 4 characters. Otherwise the year is assumed to be given by the first 2 characters. The string is interpreted from left to right to find year, month, day, hour, minute and second values, for as many parts as are present in the string. This means you should not use strings that have fewer than 6 characters. For example, if you specify '9903', thinking that will represent March, 1999, you will find that **MySQL** inserts a "zero" date into your table. This is because the year and month values are 99 and 03, but the day part is missing (zero), so the value is not a legal date.

TIMESTAMP columns store legal values using the full precision with which the value was specified, regardless of the display size. This has several implications:

- Always specify year, month, and day, even if your column types are TIMESTAMP(4) or TIMESTAMP(2). Otherwise, the value will not be a legal date and 0 will be stored.

- If you use `ALTER TABLE` to widen a narrow `TIMESTAMP` column, information will be displayed that previously was "hidden".
- Similarly, narrowing a `TIMESTAMP` column does not cause information to be lost, except in the sense that less information is shown when the values are displayed.
- Although `TIMESTAMP` values are stored to full precision, the only function that operates directly on the underlying stored value is `UNIX_TIMESTAMP()`. Other functions operate on the formatted retrieved value. This means you cannot use functions such as `HOUR()` or `SECOND()` unless the relevant part of the `TIMESTAMP` value is included in the formatted value. For example, the `HH` part of a `TIMESTAMP` column is not displayed unless the display size is at least 10, so trying to use `HOUR()` on shorter `TIMESTAMP` values produces a meaningless result.

You can to some extent assign values of one date type to an object of a different date type. However, there may be some alteration of the value or loss of information:

- If you assign a `DATE` value to a `DATETIME` or `TIMESTAMP` object, the time part of the resulting value is set to `'00:00:00'`, because the `DATE` value contains no time information.
- If you assign a `DATETIME` or `TIMESTAMP` value to a `DATE` object, the time part of the resulting value is deleted, because the `DATE` type stores no time information.
- Remember that although `DATETIME`, `DATE` and `TIMESTAMP` values all can be specified using the same set of formats, the types do not all have the same range of values. For example, `TIMESTAMP` values cannot be earlier than 1970 or later than 2037. This means that a date such as `'1968-01-01'`, while legal as a `DATETIME` or `DATE` value, is not a valid `TIMESTAMP` value and will be converted to 0 if assigned to such an object.

Be aware of certain pitfalls when specifying date values:

- The relaxed format allowed for values specified as strings can be deceiving. For example, a value such as `'10:11:12'` might look like a time value because of the ``:`` delimiter, but if used in a date context will be interpreted as the year `'2010-11-12'`. The value `'10:45:15'` will be converted to `'0000-00-00'` because `'45'` is not a legal month.
- Year values specified as two digits are ambiguous, since the century is unknown. **MySQL** interprets 2-digit year values using the following rules:
  - Year values in the range 00-69 are converted to 2000-2069.
  - Year values in the range 70-99 are converted to 1970-1999.

## The `TIME` type

**MySQL** retrieves and displays `TIME` values in `'HH:MM:SS'` format (or `'HHH:MM:SS'` format for large hours values). `TIME` values may range from `'-838:59:59'` to `'838:59:59'`. The reason the hours part may be so large is that the `TIME` type may be used not only to represent a time of day (which must be less than 24 hours), but also elapsed time or a time interval between two events (which may be much greater than 24 hours, or even negative).

You can specify `TIME` values in a variety of formats:

- As a string in `'HH:MM:SS'` format. A "relaxed" syntax is allowed--any non-numeric character may be used as the delimiter between time parts. For example, `'10:11:12'` and `'10.11.12'` are equivalent.
- As a string with no delimiters in `'HHMMSS'` format, provided that it makes sense as a time. For

example, '101112' is understood as '10:11:12', but '109712' is illegal (it has a nonsensical minute part) and becomes '00:00:00'.

- As a number in HHMMSS format, provided that it makes sense as a time. For example, 101112 is understood as '10:11:12'.
- As the result of a function that returns a value that is acceptable in a TIME context, such as CURRENT\_TIME.

For TIME values specified as strings that include a time part delimiter, it is not necessary to specify two digits for hours, minutes or seconds values that are less than 10. '8:3:2' is the same as '08:03:02'.

Be careful about assigning "short" TIME values to a TIME column. MySQL interprets values using the assumption that the rightmost digits represent seconds. (MySQL interprets TIME values as elapsed time, rather than as time of day.) For example, you might think of '11:12', '1112' and 1112 as meaning '11:12:00' (12 minutes after 11 o'clock), but MySQL interprets them as '00:11:12' (11 minutes, 12 seconds). Similarly, '12' and 12 are interpreted as '00:00:12'.

Values that lie outside the TIME range but are otherwise legal are clipped to the appropriate endpoint of the range. For example, '-850:00:00' and '850:00:00' are converted to '-838:59:59' and '838:59:59'.

Illegal TIME values are converted to '00:00:00'. Note that since '00:00:00' is itself a legal TIME value, there is no way to tell, from a value of '00:00:00' stored in a table, whether the original value was specified as '00:00:00' or whether it was illegal.

## The YEAR type

The YEAR type is a 1-byte type used for representing years.

MySQL retrieves and displays YEAR values in YYYY format. The range is 1901 to 2155.

You can specify YEAR values in a variety of formats:

- As a four-digit string in the range '1901' to '2155'.
- As a four-digit number in the range 1901 to 2155.
- As a two-digit string in the range '00' to '99'. Values in the ranges '00' to '69' and '70' to '99' are converted to YEAR values in the ranges 2000 to 2069 and 1970 to 1999.
- As a two-digit number in the range 1 to 99. Values in the ranges 1 to 69 and 70 to 99 are converted to YEAR values in the ranges 2001 to 2069 and 1970 to 1999. Note that the range for two-digit numbers is slightly different than the range for two-digit strings, since you cannot specify zero directly as a number and have it be interpreted as 2000. You *must* specify it as a string '0' or '00' or it will be interpreted as 0000.
- As the result of a function that returns a value that is acceptable in a YEAR context, such as NOW().

Illegal YEAR values are converted to 0000.

## String types

The string types are CHAR, VARCHAR, BLOB, TEXT, ENUM and SET.

## The CHAR and VARCHAR types

The CHAR and VARCHAR types are similar, but differ in the way they are stored and retrieved.

The length of a CHAR column is fixed to the length that you declare when you create the table. The length can be any value between 1 and 255. When CHAR values are stored, they are right-padded with spaces to the specified length. When CHAR values are retrieved, trailing spaces are removed.

Values in VARCHAR columns are variable-length strings. You can declare a VARCHAR column to be any length between 1 and 255, just as for CHAR columns. However, in contrast to CHAR, VARCHAR values are stored using only as many characters as are needed, plus one byte to record the length. Values are not padded; instead, trailing spaces are removed when values are stored. (This space removal differs from the ANSI SQL specification.)

If you assign a value to a CHAR or VARCHAR column that exceeds the column's maximum length, the value is truncated to fit.

The table below illustrates the differences between the two types of columns by showing the result of storing various string values into CHAR(4) and VARCHAR(4) columns:

Value	CHAR(4)	Storage required	VARCHAR(4)	Storage required
"	' '	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

The values retrieved from the CHAR(4) and VARCHAR(4) columns will be the same in each case, because trailing spaces are removed from CHAR columns upon retrieval.

Values in CHAR and VARCHAR columns are sorted and compared in case-insensitive fashion, unless the BINARY attribute was specified when the table was created. The BINARY attribute means that column values are sorted and compared in case-sensitive fashion according to the ASCII order of the machine where the MySQL server is running.

The BINARY attribute is "sticky". This means that if a column marked BINARY is used in an expression, the whole expression is compared as a BINARY value.

MySQL may silently change the type of a CHAR or VARCHAR column at table creation time. See section [Silent column specification changes](#).

## The BLOB and TEXT types

A BLOB is a binary large object that can hold a variable amount of data. The four BLOB types TINYBLOB, BLOB, MEDIUMBLOB and LONGBLOB differ only in the maximum length of the values they can hold. See section [Column type storage requirements](#).

The four TEXT types TINYTEXT, TEXT, MEDIUMTEXT and LONGTEXT correspond to the four BLOB types and

have the same maximum lengths and storage requirements. The only difference between `BLOB` and `TEXT` types is that sorting and comparison is performed in case-sensitive fashion for `BLOB` values and case-insensitive fashion for `TEXT` values. In other words, a `TEXT` is a case-insensitive `BLOB`.

If you assign a value to a `BLOB` or `TEXT` column that exceeds the column type's maximum length, the value is truncated to fit.

In most respects, you can regard a `TEXT` column as a `VARCHAR` column that can be as big as you like. Similarly, you can regard a `BLOB` column as a `VARCHAR BINARY` column. The differences are:

- `BLOB` and `TEXT` columns cannot be indexed, unlike all other **MySQL** column types.
- There is no trailing-space removal for `BLOB` and `TEXT` columns when values are stored, as there is for `VARCHAR` columns.
- `BLOB` and `TEXT` columns cannot have `DEFAULT` values.
- `BLOB` and `TEXT` columns are always `NULL` columns, even if you specify `NOT NULL` in the column specification.

**MyODBC** defines `BLOB` values as `LONGVARBINARY` and `TEXT` values as `LONGVARCHAR`.

Because `BLOB` and `TEXT` values may be extremely long, you may run up against some constraints when using them:

- If you want to use `GROUP BY` or `ORDER BY` on a `BLOB` or `TEXT` column, you must convert the column value into a fixed-length object. The standard way to do this is with the `SUBSTRING` function. For example:

```
mysql> select comment from tbl_name ORDER BY substring(comment,20);
```

If you don't do this, only the first `max_sort_length` bytes of the column are used when sorting. The default value of `max_sort_length` is 1024; this value can be changed using the `-O` option when starting the `mysqld` server. You can group on an expression involving `BLOB` or `TEXT` values by specifying the column position or by using an 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;
```

- The maximum size of a `BLOB` or `TEXT` object is determined by its type, but the largest value you can actually transmit between the client and server is determined by the amount of available memory and the size of the communications buffers. You can change the message buffer size, but you must do so on both the server and client ends. See section [10.1 Tuning server parameters](#).

Note that each `BLOB` or `TEXT` value is represented internally by a separately-allocated object. This is in contrast to all other column types, for which storage is allocated once per column when the table is opened.

## The `ENUM` type

An `ENUM` is a string object whose value normally is chosen from a list of allowed values that are enumerated explicitly in the column specification at table creation time.

The value may also be the empty string ( " ") or NULL under certain circumstances:

- If you insert an invalid value into an ENUM (that is, a string not present in the list of allowed values), the empty string is inserted instead as a special error value.
- If an ENUM is declared NULL, NULL is also a legal value for the column, and the default value is NULL. If an ENUM is declared NOT NULL, the default value is the first element of the list of allowed values.

Each enumeration value has an index:

- Values from the list of allowable elements in the column specification are numbered beginning with 1.
- The index value of the empty string error value is 0. This means that you can use the following SELECT statement to find rows into which invalid ENUM values were assigned:

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

- The index of the NULL value is NULL.

For example, a column specified as ENUM( "one", "two", "three" ) can have any of the values shown below. The index of each value is also shown:

Value	Index
NULL	NULL
" "	0
"one"	1
"two"	2
"three"	3

An enumeration can have a maximum of 65535 elements.

Lettercase is irrelevant when you assign values to an ENUM column. However, values retrieved from the column later have lettercase matching the values that were used to specify the allowable values at table creation time.

If you retrieve an ENUM in a numeric context, the column value's index is returned. If you store a number into an ENUM, the number is treated as an index, and the the value stored is the enumeration member with that index.

ENUM values are sorted according to the order in which the enumeration members were listed in the column specification. (In other words, ENUM values are sorted according to their index numbers.) For example, "a" sorts before "b" for ENUM( "a", "b" ), but "b" sorts before "a" for ENUM( "b", "a" ). The empty string sorts before non-empty strings, and NULL values sort before all other enumeration values.

If you want to get all possible values for an ENUM column, you should use: SHOW COLUMNS FROM table\_name LIKE enum\_column\_name and parse the ENUM definition in the second column.

## The SET type

A SET is a string object that can have zero or more values, each of which must be chosen from a list of allowed values specified when the table is created. SET column values that consist of multiple set members are specified with members separated by commas (`,`). A consequence of this is that SET member values cannot themselves contain commas.

For example, a column specified as SET("one", "two") NOT NULL can have any of these values:

```
" "  
"one"  
"two"  
"one,two"
```

A SET can have a maximum of 64 different members.

MySQL stores SET values numerically, with the low-order bit of the stored value corresponding to the first set member. If you retrieve a SET value in a numeric context, the value retrieved has bits set corresponding to the set members that make up the column value. If a number is stored into a SET column, the bits that are set in the binary representation of the number determine the set members in the column value. Suppose a column is specified as SET("a", "b", "c", "d"). Then the members have the following bit values:

SET member	Decimal value	Binary value
a	1	0001
b	2	0010
c	4	0100
d	8	1000

If you assign a value of 9 to this column, that is 1001 in binary, so the first and fourth SET value members "a" and "d" are selected and the resulting value is "a,d".

For a value containing more than one SET element, it does not matter what order the elements are listed in when you insert the value. It also doesn't not matter how many times a given element is listed in the value. When the value is retrieved later, each element in the value will appear once, with elements listed according to the order in which they were specified at table creation time. For example, if a column is specified as SET("a", "b", "c", "d"), then "a,d", "d,a" and "d,a,a,d,d" will all appear as "a,d" when retrieved.

SET values are sorted numerically. NULL values sort before non-NULL SET values.

Normally, you perform a SELECT on a SET column using the LIKE operator or the FIND\_IN\_SET() function:

```
mysql> SELECT * FROM tbl_name WHERE set_col LIKE '%value%';  
mysql> SELECT * FROM tbl_name WHERE FIND_IN_SET('value',set_col)>0;
```

But the following will also work:

```
mysql> SELECT * FROM tbl_name WHERE set_col = 'val1,val2';
```

```
mysql> SELECT * FROM tbl_name WHERE set_col & 1;
```

The first of these statements looks for an exact match. The second looks for values containing the first set member.

If you want to get all possible values for an SET column, you should use: `SHOW COLUMNS FROM table_name LIKE set_column_name` and parse the SET definition in the second column.

## Choosing the right type for a column

For the most efficient use of storage, try to use the most precise type in all cases. For example, if an integer column will be used for values in the range between 1 and 99999, `MEDIUMINT UNSIGNED` is the best type.

Accurate representation of monetary values is a common problem. In **MySQL**, you should use the `DECIMAL` type. This is stored as a string, so no loss of accuracy should occur. If accuracy is not too important, the `DOUBLE` type may also be good enough.

For high precision, you can always convert to a fixed-point type stored in a `BIGINT`. This allows you to do all calculations with integers and convert results back to floating-point values only when necessary.

See section [10.16 What are the different row formats? Or, when should VARCHAR/CHAR be used?](#).

## Column indexes

All **MySQL** column types can be indexed except `BLOB` and `TEXT` types. Use of indexes on the relevant columns is the best way to improve the performance of `SELECT` operations.

A table may have up to 16 indexes. The maximum index length is 256 bytes, although this may be changed when compiling **MySQL**.

You cannot index a column that may contain `NULL` values, so indexed columns must be declared `NOT NULL`.

For `CHAR` and `VARCHAR` columns, you can index a prefix of a column. This is much faster and requires less disk space than indexing the whole column. The syntax to use in the `CREATE TABLE` statement to index a column prefix looks like this:

```
KEY index_name (col_name(length))
```

The example below creates an index for the first 10 characters of the `name` column:

```
mysql> CREATE TABLE test (  
    name CHAR(200) NOT NULL,  
    KEY index_name (name(10)));
```

## Multiple-column indexes

**MySQL** can create indexes from multiple columns. An index may consist up to 15 columns (or column prefixes, for `CHAR` and `VARCHAR` columns).

A multiple-column index can be considered a sorted array containing values that are created by concatenating the values of the indexed columns.

**MySQL** uses multiple-column indexes in such a way that queries are fast when you specify a known quantity for the first column of the index in a `WHERE` clause, even if you don't specify values for the other columns.

Suppose a table is created using the following specification:

```
mysql> CREATE TABLE test (  
    id INT NOT NULL,  
    last_name CHAR(30) NOT NULL,  
    first_name CHAR(30) NOT NULL,  
    PRIMARY KEY (id),  
    INDEX name (last_name,first_name));
```

Then the index `name` is an index over `last_name` and `first_name`. The index will be used for queries that specify values in a known range for `last_name`, or for both `last_name` and `first_name`. Therefore, the `name` index will be used in the following queries:

```
mysql> SELECT * FROM test WHERE last_name="Widenius";  
  
mysql> SELECT * FROM test WHERE last_name="Widenius"  
    AND first_name="Michael";  
  
mysql> SELECT * FROM test WHERE last_name="Widenius"  
    AND (first_name="Michael" OR first_name="Monty");  
  
mysql> SELECT * FROM test WHERE last_name="Widenius"  
    AND first_name >="M" AND first_name < "N";
```

However, the `name` index will NOT be used in the following queries:

```
mysql> SELECT * FROM test WHERE first_name="Michael";  
  
mysql> SELECT * FROM test WHERE last_name="Widenius"  
    OR first_name="Michael";
```

For more information on the manner in which **MySQL** uses indexes to improve query performance, see section [10.4 How MySQL uses indexes](#).

## Using column types from other database engines

To make it easier to use code written for SQL implementations from other vendors, **MySQL** maps column types as shown in the table below. These mappings make it easier to move table definitions from other database engines to **MySQL**:

Other vendor type	MySQL type
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
MIDDLEINT	MEDIUMINT
VARBINARY (NUM)	VARCHAR (NUM) BINARY

Column type mapping occurs at table creation time. If you create a table with types used by other vendors and then issue a `DESCRIBE tbl_name` statement, **MySQL** reports the table structure using the equivalent **MySQL** types.

## Functions for use in `SELECT` and `WHERE` clauses

A `select_expression` or `where_definition` in a SQL statement can consist of any expression using the functions described below.

An expression that contains `NULL` always produces a `NULL` value unless otherwise indicated in the documentation for the operators and functions involved in the expression.

**Note:** There must be no whitespace between a function name and the parenthesis following it. This helps the **MySQL** parser distinguish between function calls and references to tables or columns that happen to have the same name as a function. Spaces around arguments are permitted, though.

For the sake of brevity, examples display the output from the `mysql` program in abbreviated form. So this:

```
mysql> select MOD(29,9);
1 rows in set (0.00 sec)
```

```
+-----+
| mod(29,9) |
+-----+
|          2 |
+-----+
```

Is displayed like this:

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

## Grouping functions

( ... )

Parentheses. Use these to force the order of evaluation in an expression.

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

## Normal arithmetic operations

The usual arithmetic operators are available. Note that in the case of -, + and \*, the result is calculated with `BIGINT` (64-bit) precision if both arguments are integers!

+ Addition

```
mysql> select 3+5;
      -> 8
```

- Subtraction

```
mysql> select 3-5;
      -> -2
```

\* Multiplication

```
mysql> select 3*5;
      -> 15
mysql> select 18014398509481984*18014398509481984.0;
      -> 324518553658426726783156020576256.0
mysql> select 18014398509481984*18014398509481984;
      -> 0
```

The result of the last expression is incorrect because the result of the integer multiplication exceeds the 64-bit range of `BIGINT` calculations.

/ Division

```
mysql> select 3/5;
      -> 0.60
```

Division by zero produces a `NULL` result:

```
mysql> select 102/(1-1);
      -> NULL
```

A division will be calculated with `BIGINT` arithmetic only if performed in a context where its result is converted to an integer!

## Bit functions

**MySQL** uses `BIGINT` (64-bit) arithmetic for bit operations, so these operators have a maximum range of 64 bits.

## | Bitwise OR

```
mysql> select 29 | 15;
-> 31
```

## & Bitwise AND

```
mysql> select 29 & 15;
-> 13
```

## << Shifts a longlong (BIGINT) number to the left.

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

## >> Shifts a longlong (BIGINT) number to the right.

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

## BIT\_COUNT(N)

Returns the number of bits that are set in the argument N.

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

## Logical operations

All logical functions return 1 (TRUE) or 0 (FALSE).

### NOT

! Logical NOT. Returns 1 if the argument is 0, otherwise returns 0. Exception: NOT NULL returns NULL.

```
mysql> select NOT 1;
-> 0
mysql> select NOT NULL;
-> NULL
mysql> select ! (1+1);
-> 0
mysql> select ! 1+1;
-> 1
```

The last example returns 1 because the expression evaluates the same way as (!1)+1.

### OR

|| Logical OR. Returns 1 if either argument is not 0 and not NULL.

```
mysql> select 1 || 0;
-> 1
mysql> select 0 || 0;
-> 0
mysql> select 1 || NULL;
-> 1
```

### AND

&& Logical AND. Returns 0 if either argument is 0 or NULL, otherwise returns 1.

```
mysql> select 1 && NULL;
-> 0
mysql> select 1 && 0;
-> 0
```

## Comparison operators

Comparison operations result in a value of 1 (TRUE), 0 (FALSE) or NULL. These functions work for both numbers and strings. Strings are automatically converted to numbers and numbers to strings as needed (as in Perl).

**MySQL** performs comparisons using the following rules:

- If one or both arguments are NULL, the result of the comparison is NULL.
- If both arguments in a comparison operation are strings, they are compared as strings.
- If both arguments are integers, they are compared as integers.
- If one of the arguments is a `TIMESTAMP` or `DATETIME` column and the other argument is a constant, the constant is converted to a timestamp before the comparison is performed. This is done to be more ODBC-friendly.
- In all other cases, the arguments are compared as floating-point (real) numbers.

By default, string comparisons are done in case-independent fashion using the current character set (ISO-8859-1 Latin1 by default, which also works excellently for English).

The examples below illustrate conversion of strings to numbers for comparison operations:

```
mysql> SELECT 1 > '6x';
-> 0
mysql> SELECT 7 > '6x';
-> 1
mysql> SELECT 0 > 'x6';
-> 0
mysql> SELECT 0 = 'x6';
-> 1
```

= Equal

```
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
```

<>

!= Not equal

```
mysql> select '.01' <> '0.01';
-> 1
mysql> select .01 <> '0.01';
-> 0
```

```
mysql> select 'zapp' <> 'zappp';
-> 1
```

**<=** Less than or equal

```
mysql> select 0.1 <= 2;
-> 1
```

**<** Less than

```
mysql> select 2 <= 2;
-> 1
```

**>=** Greater than or equal

```
mysql> select 2 >= 2;
-> 1
```

**>** Greater than

```
mysql> select 2 > 2;
-> 0
```

**<=>** Null safe equal

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

**ISNULL(expr)**

If *expr* is NULL, ISNULL() returns 1, otherwise it returns 0.

```
mysql> select ISNULL(1+1);
-> 0
mysql> select ISNULL(1/0);
-> 1
```

Note that a comparison of NULL values using = will always be false!

**expr BETWEEN min AND max**

If *expr* is greater than or equal to *min* and *expr* is less than or equal to *max*, BETWEEN returns 1, otherwise it returns 0. This is equivalent to the expression (*min* <= *expr* AND *expr* <= *max*) if all the arguments are of the same type. The first argument (*expr*) determines how the comparison is performed. If *expr* is a string expression, a case-insensitive string comparison is done. If *expr* is a binary string, a case-sensitive string comparison is done. If *expr* is an integer expression, an integer comparison is done. Otherwise, a floating-point (real) comparison is done.

```
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 IN (value,...)**

Returns 1 if *expr* is any of the values in the IN list, else returns 0. If all values are constants, then all values are evaluated according to the type of *expr* and sorted. The search for the item is then

done using a binary search. This means `IN` is very quick if the `IN` value list consists entirely of constants. If `expr` is a case-sensitive string expression, the string comparison is performed in case-sensitive fashion.

```
mysql> select 2 IN (0,3,5,'wefwf');
-> 0
mysql> select 'wefwf' IN (0,3,5,'wefwf');
-> 1
```

`expr NOT IN (value,...)`

Same as `NOT (expr IN (value,...))`.

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

Returns 0 if  $N < N1$ , 1 if  $N < N2$  and so on. All arguments are treated as numbers. It is required that  $N1 < N2 < N3 < \dots < Nn$  for this function to work correctly. This is because a binary search is used (very fast).

```
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
```

## String comparison functions

Normally, if any expression in a string comparison is case sensitive, the comparison is performed in case-sensitive fashion.

`expr1 LIKE expr2 [ESCAPE 'escape-char']`

SQL simple regular expression comparison. Returns 1 (TRUE) or 0 (FALSE). With `LIKE` you can use the following two wildcard characters:

<code>%</code>	Matches any number of characters, even zero characters
<code>_</code>	Matches exactly one character

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

To test for literal instances of a wildcard character, precede the character with the escape character. If you don't specify the `ESCAPE` character, `\\` is assumed:

<code>\\%</code>	Matches one <code>%</code> character
<code>\\_</code>	Matches one <code>_</code> character

```
mysql> select 'David!' LIKE 'David\\_';
-> 0
mysql> select 'David_' LIKE 'David\\_';
-> 1
```

To specify a different escape character, use the `ESCAPE` clause:

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

LIKE is allowed on numeric expressions! (This is a **MySQL** extension to the ANSI SQL LIKE.)

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

Note: Because **MySQL** uses the C escape syntax in strings (e.g., ``\n``), you must double any ``\`` that you use in your LIKE strings. For example, to search for ``\n``, specify it as ```\n``. To search for ````, specify it as ```\``` (the backslashes are stripped once by the parser, and another time when the pattern match is done, leaving a single backslash to be matched).

```
expr1 NOT LIKE expr2 [ESCAPE 'escape-char']
```

Same as NOT (expr1 LIKE expr2 [ESCAPE 'escape-char']).

```
expr REGEXP pat
```

```
expr RLIKE pat
```

Performs a pattern match of a string expression `expr` against a pattern `pat`. The pattern can be an extended regular expression. See section **H Description of MySQL regular expression syntax**.

Returns 1 if `expr` matches `pat`, otherwise returns 0. RLIKE is a synonym for REGEXP, provided for `mSQL` compatibility. Note: Because **MySQL** uses the C escape syntax in strings (e.g., ``\n``), you must double any ``\`` that you use in your REGEXP strings.

```
mysql> select 'Monty!' REGEXP 'm%y%';
-> 0
mysql> select 'Monty!' REGEXP '.*';
-> 1
mysql> select 'new*\n*line' REGEXP 'new\``.*\``*line';
-> 1
```

REGEXP and RLIKE use the current character set (ISO-8859-1 Latin1 by default) when deciding the type of a character.

```
expr NOT REGEXP expr
```

Same as NOT (expr REGEXP expr).

```
STRCMP(expr1,expr2)
```

STRCMP() returns 0 if the strings are the same, -1 if the first argument is smaller than the second according to the current sort order, and 1 otherwise.

```
mysql> select STRCMP('text', 'text2');
-> -1
mysql> select STRCMP('text2', 'text');
-> 1
mysql> select STRCMP('text', 'text');
-> 0
```

## Cast operators

**BINARY**

The **BINARY** operator casts the string following it to a binary string. This is an easy way to force a column comparison to be case independent even if the column isn't defined as **BINARY** or **BLOB**.

```
mysql> select "a" = "A";
-> 1
mysql> select BINARY "a" = "A";
-> 0
```

**BINARY** was introduced in **MySQL 3.23.0**

## Control flow functions

IFNULL(expr1, expr2)

If expr1 is not NULL, IFNULL() returns expr1, else it returns expr2. IFNULL() returns a numeric or string value, depending on the context in which it is used.

```
mysql> select IFNULL(1,0);
-> 1
mysql> select IFNULL(0,10);
-> 0
mysql> select IFNULL(1/0,10);
-> 10
mysql> select IFNULL(1/0,'yes');
-> 'yes'
```

IF(expr1, expr2, expr3)

If expr1 is TRUE (expr1 <> 0 and expr1 <> NULL) then IF() returns expr2, else it returns expr3. IFNULL() returns a numeric or string value, depending on the context in which it is used.

```
mysql> select IF(1>2,2,3);
-> 3
mysql> select IF(1<2,'yes','no');
-> 'yes'
mysql> select IF(strcmp('test','test1'),'yes','no');
-> 'no'
```

expr1 is evaluated as an integer value, which means that if you are testing floating-point or string values, you should do so using a comparison operation.

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

In the first case above, IF(0.1) returns 0 because 0.1 is converted to an integer value, resulting in a test of IF(0). This may not be what you expect. In the second case, the comparison tests the original floating-point value to see whether it is non-zero. The result of the comparison is used as an integer.

## Mathematical functions

All mathematical functions return NULL in case of an error.

- Unary minus. Changes the sign of the argument.

```
mysql> select - 2;
-> -2
```

Note that if this operator is used with a BIGINT, the return value is a BIGINT! This means that you should avoid using - on integers that may have the value of  $-2^{63}$ !

ABS(X)

Returns the absolute value of x.

```
mysql> select ABS(2);
```

```
mysql> select ABS(-32);
-> 2
-> 32
```

This function is safe to use with `BIGINT` values.

`SIGN(X)`

Returns the sign of the argument as -1, 0 or 1, depending on whether x is negative, zero, or positive.

```
mysql> select SIGN(-32);
-> -1
mysql> select SIGN(0);
-> 0
mysql> select SIGN(234);
-> 1
```

`MOD(N, M)`

`%` Modulo (like the `%` operator in C). Returns the remainder of N divided by M.

```
mysql> select MOD(234, 10);
-> 4
mysql> select 253 % 7;
-> 1
mysql> select MOD(29, 9);
-> 2
```

This function is safe to use with `BIGINT` values.

`FLOOR(X)`

Returns the largest integer value not greater than x.

```
mysql> select FLOOR(1.23);
-> 1
mysql> select FLOOR(-1.23);
-> -2
```

Note that the return value is converted to a `BIGINT`!

`CEILING(X)`

Returns the smallest integer value not less than x.

```
mysql> select CEILING(1.23);
-> 2
mysql> select CEILING(-1.23);
-> -1
```

Note that the return value is converted to a `BIGINT`!

`ROUND(X)`

Returns the argument x, rounded to an integer.

```
mysql> select ROUND(-1.23);
-> -1
mysql> select ROUND(-1.58);
-> -2
mysql> select ROUND(1.58);
-> 2
```

Note that the return value is converted to a `BIGINT`!

`ROUND(X, D)`

Returns the argument  $x$ , rounded to a number with  $D$  decimals. If  $D$  is 0, the result will have no decimal point or fractional part.

```
mysql> select ROUND(1.298, 1);
      -> 1.3
mysql> select ROUND(1.298, 0);
      -> 1
```

Note that the return value is converted to a `BIGINT`!

`EXP(X)`

Returns the value of  $e$  (the base of natural logarithms) raised to the power of  $x$ .

```
mysql> select EXP(2);
      -> 7.389056
mysql> select EXP(-2);
      -> 0.135335
```

`LOG(X)`

Returns the natural logarithm of  $x$ .

```
mysql> select LOG(2);
      -> 0.693147
mysql> select LOG(-2);
      -> NULL
```

If you want the log of a number  $x$  to some arbitrary base  $B$ , use the formula  $\text{LOG}(X) / \text{LOG}(B)$ .

`LOG10(X)`

Returns the base-10 logarithm of  $x$ .

```
mysql> select LOG10(2);
      -> 0.301030
mysql> select LOG10(100);
      -> 2.000000
mysql> select LOG10(-100);
      -> NULL
```

`POW(X, Y)`

`POWER(X, Y)`

Returns the value of  $x$  raised to the power of  $y$ .

```
mysql> select POW(2, 2);
      -> 4.000000
mysql> select POW(2, -2);
      -> 0.250000
```

`SQRT(X)`

Returns the non-negative square root of  $x$ .

```
mysql> select SQRT(4);
      -> 2.000000
mysql> select SQRT(20);
      -> 4.472136
```

`PI()` Returns the value of  $\pi$ .

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

#### COS(X)

Returns the cosine of x, where x is given in radians.

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

#### SIN(X)

Returns the sine of x, where x is given in radians.

```
mysql> select SIN(PI());  
-> 0.000000
```

#### TAN(X)

Returns the tangent of x, where x is given in radians.

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

#### ACOS(X)

Returns the arc cosine of x, that is, the value whose cosine is x. Returns NULL if x is not in the range -1 to 1.

```
mysql> select ACOS(1);  
-> 0.000000  
mysql> select ACOS(1.0001);  
-> NULL  
mysql> select ACOS(0);  
-> 1.570796
```

#### ASIN(X)

Returns the arc sine of x, that is, the value whose sine is x. Returns NULL if x is not in the range -1 to 1.

```
mysql> select ASIN(0.2);  
-> 0.201358  
mysql> select ASIN('foo');  
-> 0.000000
```

#### ATAN(X)

Returns the arc tangent of x, that is, the value whose tangent is x.

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

#### ATAN2(X, Y)

Returns the arc tangent of the two variables x and y. It is similar to calculating the arc tangent of y / x, except that the signs of both arguments are used to determine the quadrant of the result.

```
mysql> select ATAN(-2,2);  
-> -0.785398  
mysql> select ATAN(PI(),0);  
-> 1.570796
```

#### COT(X)

Returns the cotangent of x.

```
mysql> select COT(12);
      -> -1.57267341
mysql> select COT(0);
      -> NULL
```

RAND()

RAND(N)

Returns a random floating-point value in the range 0 to 1.0. If an integer argument N is specified, it is used as the seed value.

```
mysql> select RAND();
      -> 0.5925
mysql> select RAND(20);
      -> 0.1811
mysql> select RAND(20);
      -> 0.1811
mysql> select RAND();
      -> 0.2079
mysql> select RAND();
      -> 0.7888
```

You can't use a column with RAND() values in an ORDER BY clause, because ORDER BY would evaluate the column multiple times.

LEAST(X,Y,...)

With two or more arguments, returns the smallest (minimum-valued) argument. The arguments are compared using the following rules:

- If the return value is used in an INTEGER context, or all arguments are integer-valued, they are compared as integers.
- If the return value is used in a REAL context, or all arguments are real-valued, they are compared as reals.
- If any argument is a case-sensitive string, the arguments are compared as case-sensitive strings.
- In other cases, the arguments are compared as case-insensitive strings.

```
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"
```

In MySQL versions prior to 3.22.5, you can use MIN() instead of LEAST.

GREATEST(X,Y,...)

Returns the largest (maximum-valued) argument. The arguments are compared using the same rules as for 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"
```

In MySQL versions prior to 3.22.5, you can use MAX() instead of GREATEST.

DEGREES(X)

Returns the argument *x*, converted from radians to degrees.

```
mysql> select DEGREES(PI());
-> 180.000000
```

RADIANS(X)

Returns the argument *x*, converted from degrees to radians.

```
mysql> select RADIANS(90);
-> 1.570796
```

TRUNCATE(X,D)

Returns the number *x*, truncated to *D* decimals.

```
mysql> select TRUNCATE(1.223,1);
-> 1.2
mysql> select TRUNCATE(1.999,1);
-> 1.9
mysql> select TRUNCATE(1.999,0);
-> 1
```

## String functions

String-valued functions return `NULL` if the length of the result would be greater than the `max_allowed_packet` server parameter. See section [10.1 Tuning server parameters](#).

For functions that operate on string positions, the first position is numbered 1.

ASCII(str)

Returns the ASCII code value of the leftmost character of the string *str*. Returns 0 if *str* is the empty string. Returns `NULL` if *str* is `NULL`.

```
mysql> select ASCII('2');
-> 50
mysql> select ASCII(2);
-> 50
mysql> select ASCII('dx');
-> 100
```

CONV(N,from\_base,to\_base)

Converts numbers between different number bases. Returns a string representation of the number *N*, converted from base *from\_base* to base *to\_base*. Returns `NULL` if any argument is `NULL`. The argument *N* is interpreted as an integer, but may be specified as an integer or a string. The minimum base is 2 and the maximum base is 36. If *to\_base* is a negative number, *N* is regarded as a signed number. Otherwise, *N* is treated as unsigned. `CONV` works with 64-bit precision.

```
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"+"10"+0xa,10,10);
-> '40'
```

#### BIN(N)

Returns a string representation of the binary value of N, where N is a longlong (BIGINT) number. This is equivalent to CONV(N, 10, 2). Returns NULL if N is NULL.

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

#### OCT(N)

Returns a string representation of the octal value of N, where N is a longlong number. This is equivalent to CONV(N, 10, 8). Returns NULL if N is NULL.

```
mysql> select OCT(12);
      -> '14'
```

#### HEX(N)

Returns a string representation of the hexadecimal value of N, where N is a longlong (BIGINT) number. This is equivalent to CONV(N, 10, 16). Returns NULL if N is NULL.

```
mysql> select HEX(255);
      -> 'FF'
```

#### CHAR(N, ...)

CHAR() interprets the arguments as integers and returns a string consisting of the characters given by the ASCII code values of those integers. NULL values are skipped.

```
mysql> select CHAR(77,121,83,81,'76');
      -> 'MySQL'
mysql> select CHAR(77,77.3,'77.3');
      -> 'MMM'
```

#### CONCAT(X, Y, ...)

Returns the string that results from concatenating the arguments. Returns NULL if any argument is NULL. May have more than 2 arguments.

```
mysql> select CONCAT('My', 'S', 'QL');
      -> 'MySQL'
mysql> select CONCAT('My', NULL, 'QL');
      -> NULL
```

#### LENGTH(str)

#### OCTET\_LENGTH(str)

#### CHAR\_LENGTH(str)

#### CHARACTER\_LENGTH(str)

Returns the length of the string str.

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

#### LOCATE(substr, str)

#### POSITION(substr IN str)

Returns the position of the first occurrence of substring substr in string str. Returns 0 if substr is not in str.

```
mysql> select LOCATE('bar', 'foobarbar');
      -> 4
mysql> select LOCATE('xbar', 'foobar');
      -> 0
```

`LOCATE(substr, str, pos)`

Returns the position of the first occurrence of substring `substr` in string `str`, starting at position `pos`. Returns 0 if `substr` is not in `str`.

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

`INSTR(str, substr)`

Returns the position of the first occurrence of substring `substr` in string `str`. This is the same as the two-argument form of `LOCATE()`, except that the arguments are swapped.

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

`LPAD(str, len, padstr)`

Returns the string `str`, left-padded with the string `padstr` until `str` is `len` characters long.

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

`RPAD(str, len, padstr)`

Returns the string `str`, right-padded with the string `padstr` until `str` is `len` characters long.

```
mysql> select RPAD('hi', 5, '?');
      -> 'hi???'
```

`LEFT(str, len)`

Returns the leftmost `len` characters from the string `str`.

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

`RIGHT(str, len)`

`SUBSTRING(str FROM len)`

Returns the rightmost `len` characters from the string `str`.

```
mysql> select RIGHT('foobarbar', 4);
      -> 'rbar'
mysql> select SUBSTRING('foobarbar' FROM 4);
      -> 'rbar'
```

`SUBSTRING(str, pos, len)`

`SUBSTRING(str FROM pos FOR len)`

`MID(str, pos, len)`

Returns a substring `len` characters long from string `str`, starting at position `pos`. The variant form that uses `FROM` is ANSI SQL92 syntax.

```
mysql> select SUBSTRING('Quadratically', 5, 6);
      -> 'ratica'
```

`SUBSTRING(str, pos)`

Returns a substring from string `str` starting at position `pos`.

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

`SUBSTRING_INDEX(str, delim, count)`

Returns the substring from string `str` after `count` occurrences of the delimiter `delim`. If `count` is positive, everything to the left of the final delimiter (counting from the left) is returned. If `count` is negative, everything to the right of the final delimiter (counting from the right) is returned.

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

`LTRIM(str)`

Returns the string `str` with leading space characters removed.

```
mysql> select LTRIM('  barbar');
-> 'barbar'
```

`RTRIM(str)`

Returns the string `str` with trailing space characters removed.

```
mysql> select RTRIM('barbar  ');
-> 'barbar'
```

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

Returns the string `str` with all `remstr` prefixes and/or suffixes removed. If none of the specifiers `BOTH`, `LEADING` or `TRAILING` are given, `BOTH` is assumed. If `remstr` is not specified, spaces are removed.

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

`SOUNDEX(str)`

Returns a soundex string from `str`. Two strings that sound "about the same" should have identical soundex strings. A "standard" soundex string is 4 characters long, but the `SOUNDEX()` function returns an arbitrarily long string. You can use `SUBSTRING()` on the result to get a "standard" soundex string. All non-alphanumeric characters are ignored in the given string. All international alpha characters outside the A-Z range are treated as vowels.

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

`SPACE(N)`

Returns a string consisting of *N* space characters.

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

`REPLACE(str,from_str,to_str)`

Returns the string *str* with all occurrences of the string *from\_str* replaced by the string *to\_str*.

```
mysql> select REPLACE('www.mysql.com', 'w', 'Ww');
-> 'WwWwww.mysql.com'
```

`REPEAT(str,count)`

Returns a string consisting of the string *str* repeated *count* times. If *count*  $\leq 0$ , returns an empty string. Returns `NULL` if *str* or *count* are `NULL`.

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

`REVERSE(str)`

Returns the string *str* with the order of the characters reversed.

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

`INSERT(str,pos,len,newstr)`

Returns the string *str*, with the substring beginning at position *pos* and *len* characters long replaced by the string *newstr*.

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

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

Returns *str1* if *N* = 1, *str2* if *N* = 2, and so on. Returns `NULL` if *N* is less than 1 or greater than the number of arguments. `ELT()` is the complement of `FIELD()`.

```
mysql> select ELT(1, 'ej', 'Heja', 'hej', 'foo');
-> 'ej'
mysql> select ELT(4, 'ej', 'Heja', 'hej', 'foo');
-> 'foo'
```

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

Returns the index of *str* in the *str1, str2, str3, ...* list. Returns 0 if *str* is not found. `FIELD()` is the complement of `ELT()`.

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

`FIND_IN_SET(str,strlist)`

Returns a value 1 to *N* if the string *str* is in the list *strlist* consisting of *N* substrings. A string list is a string composed of substrings separated by ``','` characters. If the first argument is a constant string and the second is a column of type `SET`, the `FIND_IN_SET()` function is optimized to use bit arithmetic! Returns 0 if *str* is not in *strlist* or if *strlist* is the empty string. Returns

NULL if either argument is NULL. This function will not work properly if the first argument contains a `,'.

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

MAKE\_SET(bits, str1, str2, ...)

Returns a set (a string containing substrings separated by `,' characters) consisting of the strings that have the corresponding bit in bits set. str1 corresponds to bit 0, str2 to bit 1, etc. NULL strings in str1, str2, ... are not appended to the result.

```
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');
-> ''
```

LCASE(str)

LOWER(str)

Returns the string str with all characters changed to lowercase according to the current character set mapping (the default is ISO-8859-1 Latin1).

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

UCASE(str)

UPPER(str)

Returns the string str with all characters changed to uppercase according to the current character set mapping (the default is ISO-8859-1 Latin1).

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

LOAD\_FILE(file\_name)

Reads the file and returns the file contents as a string. The file must be on the server and you must specify the full pathname to the file. The file must be readable by all and be smaller than max\_allowed\_packet. If the file doesn't exist or can't be read due to one of the above reasons, the function returns NULL.

```
mysql> UPDATE table_name SET blob_column=LOAD_FILE("/tmp/picture") WHERE id=1;
```

There is no string function to convert a number to a char. There is no need for one, because **MySQL** automatically converts numbers to strings as necessary, and vice versa:

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

If a string function is given a binary string as an argument, the resulting string is also a binary string. A number converted to a string is treated as a binary string. This only affects comparisons.

## Date and time functions

See section [Date and time types](#) for a description of the range of values each type has, and the valid formats in which date and time values may be specified.

Here is an example that uses date functions. The query below selects all records with a `date_col` value from within the last 30 days:

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

`DAYOFWEEK(date)`

Returns the weekday index for `date` (1 = Sunday, 2 = Monday, ... 7 = Saturday). These index values correspond to the ODBC standard.

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

`WEEKDAY(date)`

Returns the weekday index for `date` (0 = Monday, 1 = Tuesday, ... 6 = Sunday).

```
mysql> select WEEKDAY('1997-10-04 22:23:00');
        -> 5
mysql> select WEEKDAY('1997-11-05');
        -> 2
```

`DAYOFMONTH(date)`

Returns the day of the month for `date`, in the range 1 to 31.

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

`DAYOFYEAR(date)`

Returns the day of the year for `date`, in the range 1 to 366.

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

`MONTH(date)`

Returns the month for `date`, in the range 1 to 12.

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

`DAYNAME(date)`

Returns the name of the weekday for `date`.

```
mysql> select DAYNAME("1998-02-05");
        -> 'Thursday'
```

`MONTHNAME(date)`

Returns the name of the month for `date`.

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

`QUARTER(date)`

Returns the quarter of the year for `date`, in the range 1 to 4.

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

`WEEK(date)`

`WEEK(date,first)`

With a single argument, returns the week for `date`, in the range 0 to 52, for locations where Sunday is the first day of the week. The two-argument form of `WEEK()` allows you to specify whether the week starts on Sunday or Monday. The week starts on Sunday if the second argument is 0, on Monday if the second argument is 1.

```
mysql> select WEEK('1998-02-20');
-> 7
mysql> select WEEK('1998-02-20',0);
-> 7
mysql> select WEEK('1998-02-20',1);
-> 8
```

`YEAR(date)`

Returns the year for `date`, in the range 1000 to 9999.

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

`HOURL(time)`

Returns the hour for `time`, in the range 0 to 23.

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

`MINUTE(time)`

Returns the minute for `time`, in the range 0 to 59.

```
mysql> select MINUTE('98-02-03 10:05:03');
-> 5
```

`SECOND(time)`

Returns the second for `time`, in the range 0 to 59.

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

`PERIOD_ADD(P,N)`

Adds `N` months to period `P` (in the format `YYMM` or `YYYYMM`). Returns a value in the format `YYYYMM`. Note that the period argument `P` is *not* a date value.

```
mysql> select PERIOD_ADD(9801,2);
-> 199803
```

`PERIOD_DIFF(P1,P2)`

Returns the number of months between periods `P1` and `P2`. `P1` and `P2` should be in the format `YYMM` or `YYYYMM`. Note that the period arguments `P1` and `P2` are *not* date values.

```
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)
```

These functions perform date arithmetic. They are new for **MySQL 3.22**. `ADDDATE()` and `SUBDATE()` are synonyms for `DATE_ADD()` and `DATE_SUB()`. `date` is a `DATETIME` or `DATE` value specifying the starting date. `expr` is an expression specifying the interval value to be added or subtracted from the starting date. `expr` is a string; it may start with a '-' for negative intervals. `type` is a keyword indicating how the expression should be interpreted. The following table shows how the `type` and `expr` arguments are related:

type value	Meaning	Expected expr format
SECOND	Seconds	SECONDS
MINUTE	Minutes	MINUTES
HOUR	Hours	HOURS
DAY	Days	DAYS
MONTH	Months	MONTHS
YEAR	Years	YEARS
MINUTE_SECOND	Minutes and seconds	"MINUTES:SECONDS"
HOUR_MINUTE	Hours and minutes	"HOURS:MINUTES"
DAY_HOUR	Days and hours	"DAYS HOURS"
YEAR_MONTH	Years and months	"YEARS-MONTHS"
HOUR_SECOND	Hours, minutes,	"HOURS:MINUTES:SECONDS"
DAY_MINUTE	Days, hours, minutes	"DAYS HOURS:MINUTES"
DAY_SECOND	Days, hours, minutes, seconds	"DAYS HOURS:MINUTES:SECONDS"

**MySQL** allows any non-numeric delimiter in the `expr` format. The ones shown in the table are the suggested delimiters. If the `date` argument is a `DATE` value and your calculations involve only `YEAR`, `MONTH` and `DAY` parts (that is, no time parts), the result is a `DATE` value. Otherwise the result is a `DATETIME` value.

```
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 DAY);
    -> 1998-01-01 23:59:59
mysql> select DATE_ADD("1997-12-31 23:59:59",
    INTERVAL "1:1" MINUTE_SECOND);
    -> 1998-01-01 00:01:00
mysql> select DATE_SUB("1998-01-01 00:00:00",
    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
```

If you specify an interval value that is too short (does not include all the interval parts that would

be expected from the `type` keyword), **MySQL** assumes you have left out the leftmost parts of the interval value. For example, if you specify a `type` of `DAY_SECOND`, the value of `expr` is expected to have days, hours, minutes and seconds parts. If you specify a value like "1:10", **MySQL** assumes that the days and hours parts are missing and the value represents minutes and seconds. In other words, "1:10" `DAY_SECOND` is interpreted in such a way that it is equivalent to "1:10" `MINUTE_SECOND`. This is analogous to the way that **MySQL** interprets `TIME` values as representing elapsed time rather than as time of day. If you use incorrect dates, the result is `NULL`. If you add `MONTH`, `YEAR_MONTH` or `YEAR` and the resulting date has a day that is larger than the maximum day for the new month, the day is adjusted to the maximum days in the new month.

```
mysql> select DATE_ADD('1998-01-30', Interval 1 month);
-> 1998-02-28
```

Note from the preceding example that the word `INTERVAL` and the `type` keyword are not case sensitive.

`TO_DAYS(date)`

Given a date `date`, returns a daynumber (the number of days since year 0).

```
mysql> select TO_DAYS(950501);
-> 728779
mysql> select TO_DAYS('1997-10-07');
-> 729669
```

`TO_DAYS()` is not intended for use with values that precede the advent of the Gregorian calendar (1582).

`FROM_DAYS(N)`

Given a daynumber `N`, returns a `DATE` value.

```
mysql> select FROM_DAYS(729669);
-> '1997-10-07'
```

`FROM_DAYS()` is not intended for use with values that precede the advent of the Gregorian calendar (1582).

`DATE_FORMAT(date, format)`

Formats the `date` value according to the `format` string. The following specifiers may be used in the `format` string:

%M	Month name (January..December)
%W	Weekday name (Sunday..Saturday)
%D	Day of the month with english suffix (1st, 2nd, 3rd, etc.)
%Y	Year, numeric, 4 digits
%y	Year, numeric, 2 digits
%a	Abbreviated weekday name (Sun..Sat)
%d	Day of the month, numeric (00..31)
%e	Day of the month, numeric (0..31)
%m	Month, numeric (01..12)
%c	Month, numeric (1..12)
%b	Abbreviated month name (Jan..Dec)
%j	Day of year (001..366)
%H	Hour (00..23)
%k	Hour (0..23)
%h	Hour (01..12)
%I	Hour (01..12)
%l	Hour (1..12)
%i	Minutes, numeric (00..59)
%r	Time, 12-hour (hh:mm:ss [AP]M)
%T	Time, 24-hour (hh:mm:ss)
%S	Seconds (00..59)
%s	Seconds (00..59)
%p	AM OR PM
%w	Day of the week (0=Sunday..6=Saturday)
%U	Week (0..52), where Sunday is the first day of the week.
%u	Week (0..52), where Monday is the first day of the week.
%%	Use %% to produce a literal '%'.

All other characters are just copied to the result without interpretation.

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

As of **MySQL 3.23**, the % is required before a format specifier characters. In earlier versions of **MySQL**, % was optional.

TIME\_FORMAT(time,format)

This is used like the `DATE_FORMAT()` function above, but the `format` string may contain only those format specifiers that handle hours, minutes and seconds. Other specifiers produce a `NULL` value or 0.

`CURDATE()`

`CURRENT_DATE`

Returns today's date as a value in `'YYYY-MM-DD'` or `YYYYMMDD` format, depending on whether the function is used in a string or numeric context.

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

`CURTIME()`

`CURRENT_TIME`

Returns the current time as a value in `'HH:MM:SS'` or `HHMMSS` format, depending on whether the function is used in a string or numeric context.

```
mysql> select CURTIME();
      -> '23:50:26'
mysql> select CURTIME() + 0;
      -> 235026
```

`NOW()`

`SYSDATE()`

`CURRENT_TIMESTAMP`

Returns the current date and time as a value in `'YYYY-MM-DD HH:MM:SS'` or `YYYYMMDDHHMMSS` format, depending on whether the function is used in a string or numeric context.

```
mysql> select NOW();
      -> '1997-12-15 23:50:26'
mysql> select NOW() + 0;
      -> 19971215235026
```

`UNIX_TIMESTAMP()`

`UNIX_TIMESTAMP(date)`

If called with no argument, returns a Unix timestamp (seconds since `'1970-01-01 00:00:00'` GMT). If `UNIX_TIMESTAMP()` is called with a `date` argument, it returns the value of the argument as seconds since `'1970-01-01 00:00:00'` GMT. `date` may be a `DATE` string, a `DATETIME` string, a `TIMESTAMP`, or a number in the format `YYMMDD` or `YYYYMMDD` in local time.

```
mysql> select UNIX_TIMESTAMP();
      -> 882226357
mysql> select UNIX_TIMESTAMP('1997-10-04 22:23:00');
      -> 875996580
```

When `UNIX_TIMESTAMP` is used on a `TIMESTAMP` column, the function will receive the value directly, with no implicit "string-to-unix-timestamp" conversion.

`FROM_UNIXTIME(unix_timestamp)`

Returns a representation of the `unix_timestamp` argument as a value in `'YYYY-MM-DD HH:MM:SS'` or `YYYYMMDDHHMMSS` format, depending on whether the function is used in a string or numeric context.

```
mysql> select FROM_UNIXTIME(875996580);
```

```
mysql> select FROM_UNIXTIME(875996580) + 0;
-> '1997-10-04 22:23:00'
-> 19971004222300
```

FROM\_UNIXTIME(unix\_timestamp,format)

Returns a string representation of the Unix timestamp, formatted according to the `format` string. `format` may contain the same specifiers as those listed in the entry for the `DATE_FORMAT()` function.

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

SEC\_TO\_TIME(seconds)

Returns the `seconds` argument, converted to hours, minutes and seconds, as a value in 'HH:MM:SS' or HHMMSS format, depending on whether the function is used in a string or numeric context.

```
mysql> select SEC_TO_TIME(2378);
-> '00:39:38'
mysql> select SEC_TO_TIME(2378) + 0;
-> 3938
```

TIME\_TO\_SEC(time)

Returns the `time` argument, converted to seconds.

```
mysql> select TIME_TO_SEC('22:23:00');
-> 80580
mysql> select TIME_TO_SEC('00:39:38');
-> 2378
```

## Miscellaneous functions

DATABASE()

Returns the current database name.

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

If there is no current database, `DATABASE()` returns the empty string.

USER()

SYSTEM\_USER()

SESSION\_USER()

Returns the current **MySQL** user name.

```
mysql> select USER();
-> 'davida@localhost'
```

In MySQL 3.22.11 or later, this includes the client hostname as well as the username. You can extract just the username part like this:

```
mysql> select left(USER(),instr(USER(),"@")-1);
-> 'davida'
```

PASSWORD(str)

Calculates a password string from the plaintext password `str`. This is the function that is used for encrypting **MySQL** passwords for storage in the `Password` column of the `user` grant table.

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

`PASSWORD()` encryption is non-reversible. `PASSWORD()` does not perform password encryption in the same way that Unix passwords are encrypted. You should not assume that if your Unix password and your **MySQL** password are the same, `PASSWORD()` will result in the same encrypted value as is stored in the Unix password file. See `ENCRYPT()`.

`ENCRYPT(str[,salt])`

Encrypt `str` using the Unix `crypt()` system call. The `salt` argument should be a string with 2 characters.

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

If `crypt()` is not available on your system, `ENCRYPT()` always returns `NULL`. `ENCRYPT()` ignores all but the first 8 characters of `str`, at least on some systems. This will be determined by the behavior of the underlying `crypt()` system call.

`ENCODE(str,pass_str)`

Encrypt `str` using `pass_str` as the password. To decrypt the result, use `DECODE()`. The result is a binary string. If you want to save it in a column, use a `BLOB` column type.

`DECODE(crypt_str,pass_str)`

Decrypts the encrypted string `crypt_str` using `pass_str` as the password. `crypt_str` should be a string returned from `ENCODE()`.

`LAST_INSERT_ID([expr])`

Returns the last automatically-generated value that was inserted into an `AUTO_INCREMENT` column. See section [19.4.28 `mysql\_insert\_id\(\)`](#).

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

The last ID that was generated is maintained in the server on a per-connection basis. It will not be changed by another client. It will not even be changed if you update another `AUTO_INCREMENT` column with a non-magic value (that is, a value that is not `NULL` and not 0). If `expr` is given as an argument to `LAST_INSERT_ID()` in an `UPDATE` clause, then the value of the argument is returned as a `LAST_INSERT_ID()` value. This can be used to simulate sequences: First create the table:

```
mysql> create table sequence (id int not null);
mysql> insert into sequence values (0);
```

Then the table can be used to generate sequence numbers like this:

```
mysql> update sequence set id=LAST_INSERT_ID(id+1);
```

You can generate sequences without calling `LAST_INSERT_ID()`, but the utility of using the function this way is that the ID value is maintained in the server as the last automatically-generated value. You can retrieve the new ID as you would read any normal `AUTO_INCREMENT` value in **MySQL**. For example, `LAST_INSERT_ID()` (without an argument) will return the new ID. The C API function `mysql_insert_id()` can also be used to get the value.

`FORMAT(X,D)`

Formats the number `x` to a format like `'#,###,###.##'` with `D` decimals. If `D` is 0, the result will

have no decimal point or fractional part.

```
mysql> select FORMAT(12332.1234, 2);
-> '12,332.12'
mysql> select FORMAT(12332.1,4);
-> '12,332.1000'
mysql> select FORMAT(12332.2,0);
-> '12,332'
```

VERSION()

Returns a string indicating the **MySQL** server version.

```
mysql> select VERSION();
-> '3.22.19b-log'
```

GET\_LOCK(str,timeout)

Tries to obtain a lock with a name given by the string *str*, with a timeout of *timeout* seconds. Returns 1 if the lock was obtained successfully, 0 if the attempt timed out, or NULL if an error occurred (such as running out of memory or the thread was killed with `mysqladmin kill`). A lock is released when you execute `RELEASE_LOCK()`, execute a new `GET_LOCK()` or the thread terminates. This function can be used to implement application locks or to simulate record locks.

```
mysql> select GET_LOCK("lock1",10);
-> 1
mysql> select GET_LOCK("lock2",10);
-> 1
mysql> select RELEASE_LOCK("lock2");
-> 1
mysql> select RELEASE_LOCK("lock1");
-> NULL
```

Note that the second `RELEASE_LOCK()` call returns NULL because the lock "lock1" was automatically released by the second `GET_LOCK()` call.

RELEASE\_LOCK(str)

Releases the lock named by the string *str* that was obtained with `GET_LOCK()`. Returns 1 if the lock was released, 0 if the lock wasn't locked by this thread (in which case the lock is not released) and NULL if the named lock didn't exist. The lock will not exist if it was never obtained by a call to `GET_LOCK()` or if it already has been released.

BENCHMARK(count,expr)

The `BENCHMARK()` function executes the expression *expr* repeatedly *count* times. It may be used to time how fast **MySQL** processes the expression. The result value is always 0. The intended use is in the `mysql` client, which reports query execution times.

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

The time reported is elapsed time on the client end, not CPU time on the server end. It may be advisable to execute `BENCHMARK()` several times, and interpret the result with regard to how heavily loaded the server machine is.

## Functions for use with GROUP BY clauses

If you use a group function in a statement containing no GROUP BY clause, it is equivalent to grouping on all rows.

COUNT(expr)

Returns a count of the number of non-NULL rows retrieved by a SELECT statement.

```
mysql> select student.student_name, COUNT(*)
        from student, course
        where student.student_id=course.student_id
        GROUP BY student_name;
```

COUNT(\*) is optimized to return very quickly if the SELECT retrieves from one table, no other columns are retrieved and there is no WHERE clause. For example:

```
mysql> select COUNT(*) from student;
```

AVG(expr)

Returns the average value of expr.

```
mysql> select student_name, AVG(test_score)
        from student
        GROUP BY student_name;
```

MIN(expr)

MAX(expr)

Returns the minimum or maximum value of expr. MIN() and MAX() may take a string argument; in such cases they return the minimum or maximum string value.

```
mysql> select student_name, MIN(test_score), MAX(test_score)
        from student
        GROUP BY student_name;
```

SUM(expr)

Returns the sum of expr.

STD(expr)

STDDEV(expr)

Returns the standard deviation of expr. This is an extension to ANSI SQL. The STDDEV() form of this function is provided for Oracle compatibility.

BIT\_OR(expr)

Returns the bitwise OR of all bits in expr. The calculation is performed with 64-bit (BIGINT) precision.

BIT\_AND(expr)

Returns the bitwise AND of all bits in expr. The calculation is performed with 64-bit (BIGINT) precision.

**MySQL** has extended the use of GROUP BY. You can use columns or calculations in the SELECT expressions which don't appear in the GROUP BY part. This stands for *any possible value for this group*. You can use this to get better performance by avoiding sorting and grouping on unnecessary items. For example, you don't need to group on customer.name in the following query:

```
mysql> select order.custid, customer.name, max(payments)
        from order, customer
        where order.custid = customer.custid
        GROUP BY order.custid;
```

In ANSI SQL, you would have to add `customer.name` to the `GROUP BY` clause. In **MySQL**, the name is redundant.

Don't use this feature if the columns you omit from the `GROUP BY` part aren't unique in the group!

In some cases, you can use `MIN()` and `MAX()` to obtain a specific column value even if it isn't unique. The following gives the value of `column` from the row containing the smallest value in the `sort` column:

```
substr(MIN(concat(sort, space(6-length(sort))), column), 7, length(column))
```

Note that you can't yet use expressions in `GROUP BY` or `ORDER BY` clauses. You can work around this limitation by using an alias for the expression:

```
mysql> select id, floor(value/100) as val from tbl_name
        GROUP BY id, val ORDER BY val;
```

## CREATE DATABASE syntax

```
CREATE DATABASE db_name
```

`CREATE DATABASE` creates a database with the given name. Rules for allowable database names are given in section [Database, table, index, column and alias names](#). An error occurs if the database already exists.

Databases in **MySQL** are implemented as directories containing files that correspond to tables in the database. Since there are no tables in a database when it is initially created, the `CREATE DATABASE` statement only creates a directory under the **MySQL** data directory.

You can also create databases with `mysqladmin`. See section [12.1 Overview of the different MySQL programs](#).

## DROP DATABASE syntax

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

`DROP DATABASE` drops all tables in the database and deletes the database. **Be VERY careful with this command!**

`DROP DATABASE` returns the number of files that were removed from the database directory. Normally, this is three times the number of tables, since each table corresponds to a ``.ISD'` file, a ``.ISM'` file and a ``.frm'` file.

In **MySQL** 3.22 or later, you can use the keywords `IF EXISTS` to prevent an error from occurring if the database doesn't exist.

You can also drop databases with `mysqladmin`. See section [12.1 Overview of the different MySQL programs](#).

## CREATE TABLE syntax

```
CREATE TABLE [IF NOT EXISTS] tbl_name (create_definition,...) [table_options] [selec
```

create\_definition:

```
col_name type [NOT NULL | NULL] [DEFAULT default_value] [AUTO_INCREMENT]
    [PRIMARY KEY] [reference_definition]
or    PRIMARY KEY (index_col_name,...)
or    KEY [index_name] KEY(index_col_name,...)
or    INDEX [index_name] (index_col_name,...)
or    UNIQUE [INDEX] [index_name] (index_col_name,...)
or    [CONSTRAINT symbol] FOREIGN KEY index_name (index_col_name,...)
        [reference_definition]
or    CHECK (expr)
```

type:

```
TINYINT[(length)] [UNSIGNED] [ZEROFILL]
or    SMALLINT[(length)] [UNSIGNED] [ZEROFILL]
or    MEDIUMINT[(length)] [UNSIGNED] [ZEROFILL]
or    INT[(length)] [UNSIGNED] [ZEROFILL]
or    INTEGER[(length)] [UNSIGNED] [ZEROFILL]
or    BIGINT[(length)] [UNSIGNED] [ZEROFILL]
or    REAL[(length,decimals)] [UNSIGNED] [ZEROFILL]
or    DOUBLE[(length,decimals)] [UNSIGNED] [ZEROFILL]
or    FLOAT[(length,decimals)] [UNSIGNED] [ZEROFILL]
or    DECIMAL(length,decimals) [UNSIGNED] [ZEROFILL]
or    NUMERIC(length,decimals) [UNSIGNED] [ZEROFILL]
or    CHAR(length) [BINARY]
or    VARCHAR(length) [BINARY]
or    DATE
or    TIME
or    TIMESTAMP
or    DATETIME
or    TINYBLOB
or    BLOB
or    MEDIUMBLOB
or    LONGBLOB
or    TINYTEXT
or    TEXT
or    MEDIUMTEXT
or    LONGTEXT
or    ENUM(value1,value2,value3,...)
or    SET(value1,value2,value3,...)
```

index\_col\_name:

```
col_name [(length)]
```

reference\_definition:

```
REFERENCES tbl_name [(index_col_name,...)]
    [MATCH FULL | MATCH PARTIAL]
    [ON DELETE reference_option]
    [ON UPDATE reference_option]
```

reference\_option:

```
RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT
```

table\_options:

```
type = [ISAM | MYISAM | HEAP]
or    max_rows = #
or    min_rows = #
or    avg_row_length = #
or    comment = "string"
```

```
or      auto_increment = #
```

```
select_statement:  
    [ | IGNORE | REPLACE] SELECT ... (Some legal select statement)
```

CREATE TABLE creates a table with the given name in the current database. Rules for allowable table names are given in section [Database, table, index, column and alias names](#). An error occurs if there is no current database or if the table already exists.

In **MySQL** 3.22 or later, the table name can be specified as `db_name.tbl_name`. This works whether or not there is a current database.

In **MySQL** 3.23 or later, you can use the keywords `IF NOT EXISTS` so that an error does not occur if the table already exists. Note that there is no verification that the table structures are identical.

Each table `tbl_name` is represented by three files in the database directory:

File	Purpose
<code>tbl_name.frm</code>	Table definition (form) file
<code>tbl_name.ISD</code>	Data file
<code>tbl_name.ISM</code>	Index file

For more information on the properties of the various column types, see section [Column types](#).

- If neither `NULL` nor `NOT NULL` is specified, the column is treated as though `NULL` had been specified.
- `BLOB` and `TEXT` columns are always `NULL` columns, even if you specify `NOT NULL` in the column specification.
- An integer column may have the additional attribute `AUTO_INCREMENT`. When you insert a value of `NULL` (recommended) or `0` into an `AUTO_INCREMENT` column, the column is set to `value+1`, where `value` is the largest value for the column currently in the table. `AUTO_INCREMENT` sequences begin with 1. See section [19.4.28 mysql\\_insert\\_id\(\)](#). If you delete the row containing the maximum value for an `AUTO_INCREMENT` column, the value will be reused. If you delete all rows in the table, the sequence starts over. **Note:** There can be only one `AUTO_INCREMENT` column per table, and it must be indexed. To make **MySQL** compatible with some ODBC applications, you can find the last inserted row with the following query:

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

- `NULL` values are handled differently for `TIMESTAMP` columns than for other column types. You cannot store a literal `NULL` in a `TIMESTAMP` column; setting the column to `NULL` sets it to the current date and time. Because `TIMESTAMP` columns behave this way, the `NULL` and `NOT NULL` attributes do not apply in the normal way and are ignored if you specify them. On the other hand, to make it easier for **MySQL** clients to use `TIMESTAMP` columns, the server reports that such columns may be assigned `NULL` values (which is true), even though `TIMESTAMP` never actually will contain a `NULL` value. You can see this when you use `DESCRIBE tbl_name` to get a description of your table. Note that setting a `TIMESTAMP` column to `0` is not the same as setting it to `NULL`, because `0` is a valid `TIMESTAMP` value.
- If no `DEFAULT` value is specified for a column, **MySQL** automatically assigns one. If the column

may take `NULL` as a value, the default value is `NULL`. If the column is declared as `NOT NULL`, the default value depends on the column type:

- For numeric types other than those declared with the `AUTO_INCREMENT` attribute, the default is 0. For an `AUTO_INCREMENT` column, the default value is the next value in the sequence.
- For date and time types other than `TIMESTAMP`, the default is the appropriate "zero" value for the type. For the first `TIMESTAMP` column in a table, the default value is the current date and time. See section [Date and time types](#).
- For string types other than `ENUM`, the default is the empty string. For `ENUM`, the default is the first enumeration value.
- `KEY` is a synonym for `INDEX`.
- In **MySQL**, a `UNIQUE` key can have only distinct values. An error occurs if you try to add a new row with a key that matches an existing row.
- In **MySQL** a `PRIMARY KEY` is the same thing as a unique `KEY` that is named `PRIMARY`. A table can have only one `PRIMARY KEY`. If you don't have a `PRIMARY KEY` and some applications ask for the `PRIMARY KEY` in your tables, **MySQL** will return the first `UNIQUE` key as the `PRIMARY KEY`.
- A `PRIMARY KEY` can be a multiple-column index. However, you cannot create a multiple-column index using the `PRIMARY KEY` key attribute in a column specification. Doing so will mark only that single column as primary. You must use the `PRIMARY KEY(index_col_name, ...)` syntax.
- If you don't assign a name to an index, the index will be assigned the same name as the first `index_col_name`, with an optional suffix (`_2`, `_3`, ...) to make it unique. You can see index names for a table using `SHOW INDEX FROM tbl_name`. See section [SHOW syntax \(Get information about tables, columns,...\)](#).
- Columns that are indexed cannot have `NULL` values. You must declare such columns `NOT NULL` or an error results.
- `BLOB` and `TEXT` columns cannot be indexed.
- With `col_name(length)` syntax, you can specify an index which uses only a part of a `CHAR` or `VARCHAR` column. This can make the index file much smaller. See section [Column indexes](#).
- When you use `ORDER BY` or `GROUP BY` with a `TEXT` or `BLOB` column, only the first `max_sort_length` bytes are used. See section [The BLOB and TEXT types](#).
- The `FOREIGN KEY`, `CHECK` and `REFERENCES` clauses don't actually do anything. The syntax for them is provided only for compatibility, to make it easier to port code from other SQL servers and to run applications that create tables with references. See section [5.3 Functionality missing from MySQL](#).
- Each `NULL` column takes one bit extra, rounded up to the nearest byte.
- The maximum record length in bytes can be calculated as follows:

```
row length = 1
             + (sum of column lengths)
             + (number of NULL columns + 7)/8
             + (number of variable-length columns)
```

- The `table_options` and `SELECT options` is only implemented in **MySQL** 3.23 and above.. The different table types are:

ISAM	The original table handler
MYISAM	The new binary portable table handler
HEAP	The data for this table is only stored in memory

See section [10.17 MySQL table types](#). The different table options are used to optimize the behavior of the table. In most cases, you don't have to specify these:

max_rows	Max number of rows you plan to store in the table.
min_rows	Minimum number of rows you plan to store in the table.
avg_row_length	An approximation of the average row length for the table.
comment \$tab A 60 character comment for your table	
auto_increment	The next auto_increment value you want to set for your table.

When you use a MYISAM table, **MySQL** uses the product of `max_rows * avg_row_length` to decide how big the resulting table will be. If you don't specify any of the above options, the maximum size for a table will be 4G (or 2G if your operating systems only supports 2G tables).

- If you specify a `SELECT` after the `CREATE TABLE` statement, **MySQL** will create new fields for all elements in the `SELECT`. For example:

```
mysql> CREATE TABLE test (a int not null auto_increment, primary key (a), key(b)
TYPE=HEAP SELECT b,c from test2;
```

This will create a `HEAP` table with 3 columns. Note that the table will automatically be deleted if any errors occur while copying data into the table.

## Silent column specification changes

In some cases, **MySQL** silently changes a column specification from that given in a `CREATE TABLE` statement. (This may also occur with `ALTER TABLE`.)

- `VARCHAR` columns with a length less than four are changed to `CHAR`.
- If any column in a table has a variable length, the entire row is variable-length as a result. Therefore, if a table contains any variable-length columns (`VARCHAR`, `TEXT` or `BLOB`), all `CHAR` columns longer than three characters are changed to `VARCHAR` columns. This doesn't affect how you use the columns in any way; in **MySQL**, `VARCHAR` is just a different way to store characters. **MySQL** performs this conversion because it saves space and makes table operations faster. See section [10.16 What are the different row formats? Or, when should VARCHAR/CHAR be used?](#).
- `TIMESTAMP` display sizes must be even and in the range from 2 to 14. If you specify a display size of 0 or greater than 14, the size is coerced to 14. Odd-valued sizes in the range from 1 to 13 are coerced to the next higher even number.
- You cannot store a literal `NULL` in a `TIMESTAMP` column; setting it to `NULL` sets it to the current date and time. Because `TIMESTAMP` columns behave this way, the `NULL` and `NOT NULL` attributes do not apply in the normal way and are ignored if you specify them. `DESCRIBE tbl_name` always reports that a `TIMESTAMP` column may be assigned `NULL` values.
- `BLOB` and `TEXT` columns are always `NULL` columns, even if you specify `NOT NULL` in the column specification.
- **MySQL** maps certain column types used by other SQL database vendors to **MySQL** types. See section [Using column types from other database engines](#).

If you want to see whether or not **MySQL** used a column type other than the one you specified, issue a `DESCRIBE tbl_name` statement after creating or altering your table.

Certain other column type changes may occur if you compress a table using `pack_isam`. See section [10.16 What are the different row formats? Or, when should VARCHAR/CHAR be used?](#).

## ALTER TABLE syntax

```
ALTER [IGNORE] TABLE tbl_name alter_spec [, alter_spec ...]
```

alter\_specification:

```
    ADD [COLUMN] create_definition [FIRST | AFTER column_name ]
or    ADD INDEX [index_name] (index_col_name,...)
or    ADD PRIMARY KEY (index_col_name,...)
or    ADD UNIQUE [index_name] (index_col_name,...)
or    ALTER [COLUMN] col_name {SET DEFAULT literal | DROP DEFAULT}
or    CHANGE [COLUMN] old_col_name create_definition
or    MODIFY [COLUMN] create_definition
or    DROP [COLUMN] col_name
or    DROP PRIMARY KEY
or    DROP INDEX key_name
or    RENAME [AS] new_tbl_name
or    table_option
```

`ALTER TABLE` allows you to change the structure of an existing table. For example, you can add or delete columns, create or destroy indexes, change the type of existing columns, or rename columns or the table itself. You can also change the comment for the table and type of the table. See section [CREATE TABLE syntax](#).

If you use `ALTER TABLE` to change a column specification but `DESCRIBE tbl_name` indicates that your column was not changed, it is possible that **MySQL** ignored your modification for one of the reasons described in section [Silent column specification changes](#). For example, if you try to change a `VARCHAR` column to `CHAR`, **MySQL** will still use `VARCHAR` if the table contains other variable-length columns.

`ALTER TABLE` works by making a temporary copy of the original table. The alteration is performed on the copy, then the original table is deleted and the new one is renamed. This is done in such a way that all updates are automatically redirected to the new table without any failed updates. While `ALTER TABLE` is executing, the original table is readable by other clients. Updates and writes to the table are stalled until the new table is ready.

- To use `ALTER TABLE`, you need **select**, **insert**, **delete**, **update**, **create** and **drop** privileges on the table.
- `IGNORE` is a **MySQL** extension to ANSI SQL92. It controls how `ALTER TABLE` works if there are duplicates on unique keys in the new table. If `IGNORE` isn't specified, the copy is aborted and rolled back. If `IGNORE` is specified, then for rows with duplicates on a unique key, only the first row is used; the others are deleted.
- You can issue multiple `ADD`, `ALTER`, `DROP` and `CHANGE` clauses in a single `ALTER TABLE` statement. This is a **MySQL** extension to ANSI SQL92, which allows only one of each clause per `ALTER TABLE` statement.
- `CHANGE col_name`, `DROP col_name` and `DROP INDEX` are **MySQL** extensions to ANSI SQL92.
- `MODIFY` is an Oracle extension to `ALTER TABLE`.
- The optional word `COLUMN` is a pure noise word and can be omitted.
- If you use `ALTER TABLE tbl_name RENAME AS new_name` without any other options, **MySQL** simply renames the files that correspond to the table `tbl_name`. There is no need to create the temporary table.

- `create_definition` clauses use the same syntax for `ADD` and `CHANGE` as for `CREATE TABLE`. Note that this syntax includes the column name, not just the column type. See section [CREATE TABLE syntax](#).
- You can rename a column using a `CHANGE old_col_name create_definition` clause. To do so, specify the old and new column names and the type that the column currently has. For example, to rename an `INTEGER` column from `a` to `b`, you can do this:

```
mysql> ALTER TABLE t1 CHANGE a b INTEGER;
```

If you want to change a column's type but not the name, `CHANGE` syntax still requires two column names even if they are the same. For example:

```
mysql> ALTER TABLE t1 CHANGE b b BIGINT NOT NULL;
```

However, as of **MySQL 3.22.16a**, you can also use `MODIFY` to change a column's type without renaming it:

```
mysql> ALTER TABLE t1 MODIFY b BIGINT NOT NULL;
```

- If you use `CHANGE` or `MODIFY` to shorten a column for which an index exists on part of the column (for instance, if you have an index on the first 10 characters of a `VARCHAR` column), you cannot make the column shorter than the number of characters that are indexed.
- When you change a column type using `CHANGE` or `MODIFY`, **MySQL** tries to convert data to the new type as well as possible.
- In **MySQL 3.22** or later, you can use `FIRST OR ADD ... AFTER col_name` to add a column at a specific position within a table row. The default is to add the column last.
- `ALTER COLUMN` specifies a new default value for a column or removes the old default value. If the old default is removed and the column can be `NULL`, the new default is `NULL`. If the column cannot be `NULL`, **MySQL** assigns a default value. Default value assignment is described in section [CREATE TABLE syntax](#).
- `DROP INDEX` removes an index. This is a **MySQL** extension to ANSI SQL92.
- If columns are dropped from a table, the columns are also removed from any index of which they are a part. If all columns that make up an index are dropped, the index is dropped as well.
- `DROP PRIMARY KEY` drops the primary index. If no such index exists, it drops the first `UNIQUE` index in the table. (**MySQL** marks the first `UNIQUE` key as the `PRIMARY KEY` if no `PRIMARY KEY` was specified explicitly.)
- With the C API function `mysql_info()`, you can find out how many records were copied, and (when `IGNORE` is used) how many records were deleted due to duplication of unique key values.
- The `FOREIGN KEY`, `CHECK` and `REFERENCES` clauses don't actually do anything. The syntax for them is provided only for compatibility, to make it easier to port code from other SQL servers and to run applications that create tables with references. See section [5.3 Functionality missing from MySQL](#).

Here is an example that shows some of the uses of `ALTER TABLE`. We begin with a table `t1` that is created as shown below:

```
mysql> CREATE TABLE t1 (a INTEGER,b CHAR(10));
```

To rename the table from `t1` to `t2`:

```
mysql> ALTER TABLE t1 RENAME t2;
```

To change column `a` from `INTEGER` to `TINYINT NOT NULL` (leaving the name the same), and to change column `b` from `CHAR(10)` to `CHAR(20)` as well as renaming it from `b` to `c`:

```
mysql> ALTER TABLE t2 MODIFY a TINYINT NOT NULL, CHANGE b c CHAR(20);
```

To add a new `TIMESTAMP` column named `d`:

```
mysql> ALTER TABLE t2 ADD d TIMESTAMP;
```

To add an index on column `d`, and make column `a` the primary key:

```
mysql> ALTER TABLE t2 ADD INDEX (d), ADD PRIMARY KEY (a);
```

To remove column `c`:

```
mysql> ALTER TABLE t2 DROP COLUMN c;
```

To add a new `AUTO_INCREMENT` integer column named `c`:

```
mysql> ALTER TABLE t2 ADD c INT UNSIGNED NOT NULL AUTO_INCREMENT,  
      ADD INDEX (c);
```

Note that we indexed `c`, because `AUTO_INCREMENT` columns must be indexed, and also that we declare `c` as `NOT NULL`, because indexed columns cannot be `NULL`.

## OPTIMIZE TABLE syntax

```
OPTIMIZE TABLE tbl_name
```

`OPTIMIZE TABLE` should be used if you have deleted a large part of a table or if you have made many changes to a table with variable-length rows (tables that have `VARCHAR`, `BLOB` or `TEXT` columns). Deleted records are maintained in a linked list and subsequent `INSERT` operations reuse old record positions. You can use `OPTIMIZE TABLE` to reclaim the unused space.

`OPTIMIZE TABLE` works by making a temporary copy of the original table. The old table is copied to the new table (without the unused rows), then the original table is deleted and the new one is renamed. This is done in such a way that all updates are automatically redirected to the new table without any failed updates. While `OPTIMIZE TABLE` is executing, the original table is readable by other clients. Updates and writes to the table are stalled until the new table is ready.

## DROP TABLE syntax

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

`DROP TABLE` removes one or more tables. All table data and the table definition are *removed*, so **be careful** with this command!

In **MySQL 3.22** or later, you can use the keywords `IF EXISTS` to prevent an error from occurring for tables that don't exist.

## DELETE syntax

```
DELETE [LOW_PRIORITY] FROM tbl_name
    [WHERE where_definition] [LIMIT rows]
```

DELETE deletes rows from `tbl_name` that satisfy the condition given by `where_definition`, and returns the number of records deleted.

If you issue a DELETE with no WHERE clause, all rows are deleted. **MySQL** does this by recreating the table as an empty table, which is much faster than deleting each row. In this case, DELETE returns zero as the number of affected records. (**MySQL** can't return the number of rows that were actually deleted, since the recreate is done without opening the data files. As long as the table definition file ``tbl_name.frm'` is valid, the table can be recreated this way, even if the data or index files have become corrupted.).

If you really want to know how many records are deleted when you are deleting all rows, and are willing to suffer a speed penalty, you can use a DELETE statement of this form:

```
mysql> DELETE FROM tbl_name WHERE 1>0;
```

Note that this is MUCH slower than DELETE FROM `tbl_name` with no WHERE clause, because it deletes rows one at a time.

If you specify the keyword `LOW_PRIORITY`, execution of the DELETE is delayed until no other clients are reading from the table.

Deleted records are maintained in a linked list and subsequent INSERT operations reuse old record positions. To reclaim unused space and reduce file sizes, use the OPTIMIZE TABLE statement or the `isamchk` utility to reorganize tables. OPTIMIZE TABLE is easier, but `isamchk` is faster. See section [OPTIMIZE TABLE syntax](#), and section [13.4.3 Table optimization](#).

The **MySQL**-specific `LIMIT rows` option to DELETE tells the server the maximum number of rows to be deleted before control is returned to the client. This can be used to ensure that a specific DELETE command doesn't take too much time. You can simply repeat the DELETE command until the number of affected rows is less than the LIMIT value.

## SELECT syntax

```
SELECT [STRAIGHT_JOIN] [SQL_SMALL_RESULT] [DISTINCT | ALL]
    select_expression,...
    [INTO OUTFILE 'file_name' export_options]
    [FROM table_references
        [WHERE where_definition]
        [GROUP BY col_name,...]
        [HAVING where_definition]
        [ORDER BY {unsigned_integer | col_name} [ASC | DESC] ,... ]
        [LIMIT [offset,] rows]
        [PROCEDURE procedure_name] ]
```

SELECT is used to retrieve rows selected from one or more tables. `select_expression` indicates the columns you want to retrieve. SELECT may also be used to retrieve rows computed without reference to any table. For example:

```
mysql> SELECT 1 + 1;
```

-> 2

All keywords used must be given in exactly the order shown above. For example, a `HAVING` clause must come after any `GROUP BY` clause and before any `ORDER BY` clause.

- A `SELECT` expression may be given an alias using `AS`. The alias is used as the expression's column name and can be used with `ORDER BY` or `HAVING` clauses. For example:

```
mysql> select concat(last_name,', ',first_name) AS full_name
        from mytable ORDER BY full_name;
```

- The `FROM table_references` clause indicates the tables from which to retrieve rows. If you name more than one table, you are performing a join. For information on join syntax, see section [JOIN syntax](#).
- You can refer to a column as `col_name`, `tbl_name.col_name` or `db_name.tbl_name.col_name`. You need not specify a `tbl_name` or `db_name.tbl_name` prefix for a column reference in a `SELECT` statement unless the reference would be ambiguous. See section [Database, table, index, column and alias names](#), for examples of ambiguity that require the more explicit column reference forms.
- A table reference may be aliased using `tbl_name [AS] alias_name`.

```
mysql> select t1.name, t2.salary from employee AS t1, info AS t2
        where t1.name = t2.name;
mysql> select t1.name, t2.salary from employee t1, info t2
        where t1.name = t2.name;
```

- Columns selected for output may be referred to in `ORDER BY` and `GROUP BY` clauses using column names, column aliases or column positions. Column positions begin with 1.

```
mysql> select college, region, seed from tournament
        ORDER BY region, seed;
mysql> select college, region AS r, seed AS s from tournament
        ORDER BY r, s;
mysql> select college, region, seed from tournament
        ORDER BY 2, 3;
```

To sort in reverse order, add the `DESC` (descending) keyword to the name of the column in the `ORDER BY` clause that you are sorting by. The default is ascending order; this may be specified explicitly using the `ASC` keyword.

- The `HAVING` clause can refer to any column or alias named in the `select_expression`. It is applied last, just before items are sent to the client, with no optimization. Don't use `HAVING` for items that should be in the `WHERE` clause. For example, do not write this:

```
mysql> select col_name from tbl_name HAVING col_name > 0;
```

Write this instead:

```
mysql> select col_name from tbl_name WHERE col_name > 0;
```

In **MySQL 3.22.5** or later, you can also write queries like this:

```
mysql> select user,max(salary) from users
        group by user HAVING max(salary)>10;
```

In older **MySQL** versions, you can write this instead:

```
mysql> select user,max(salary) AS sum from users
      group by user HAVING sum>10;
```

- `STRAIGHT_JOIN` forces the optimizer to join the tables in the order in which they are listed in the `FROM` clause. You can use this to speed up a query if the optimizer joins the tables in non-optimal order. See section [EXPLAIN syntax \(Get information about a SELECT\)](#).
- `SQL_SMALL_RESULT` can be used with `GROUP BY` or `DISTINCT` to tell the optimizer that the result set will be small. In this case, **MySQL** will use fast temporary tables to store the resulting table instead of using sorting. `SQL_SMALL_RESULT` is a **MySQL** extension to ANSI SQL92.
- The `LIMIT` clause can be used to constrain the number of rows returned by the `SELECT` statement. `LIMIT` takes one or two numeric arguments. If two arguments are given, the first specifies the offset of the first row to return, the second specifies the maximum number of rows to return. The offset of the initial row is 0 (not 1).

```
mysql> select * from table LIMIT 5,10; # Retrieve rows 6-15
```

If one argument is given, it indicates the maximum number of rows to return.

```
mysql> select * from table LIMIT 5; # Retrieve first 5 rows
```

In other words, `LIMIT n` is equivalent to `LIMIT 0,n`.

- The `SELECT ... INTO OUTFILE 'file_name'` form of `SELECT` writes the selected rows to a file. The file is created on the server host, and cannot already exist (among other things, this prevents database tables and files such as `/etc/passwd` from being destroyed). You must have the **file** privilege on the server host to use this form of `SELECT`. `SELECT ... INTO OUTFILE` is the complement of `LOAD DATA INFILE`; the syntax for the `export_options` part of the statement consists of the same `FIELDS` and `LINES` clauses that are used with the `LOAD DATA INFILE` statement. See section [LOAD DATA INFILE syntax](#). In the resulting text file, only the following characters are escaped by the `ESCAPED BY` character:
  - The `ESCAPED BY` character
  - The first character in `FIELDS TERMINATED BY`
  - The first character in `LINES TERMINATED BY`

Additionally `ASCII 0` is converted to `ESCAPED BY` followed by `0` (`ASCII 48`). The reason for the above is that one **MUST** escape any `FIELDS TERMINATED BY`, `ESCAPED BY` or `LINES TERMINATED BY` characters to be able to reliably be able to read the file back. `ASCII 0` is escaped to make it easier to view with some pages. As the above file doesn't have to conform to any SQL syntax nothing else needs to be escaped. As the resulting file doesn't have to conform to the SQL syntax nothing else needs to be escaped.

## JOIN syntax

**MySQL** supports the following `JOIN` syntaxes for use in `SELECT` statements:

```
table_reference, table_reference
table_reference [CROSS] JOIN table_reference
table_reference STRAIGHT_JOIN table_reference
table_reference LEFT [OUTER] JOIN table_reference ON conditional_expr
table_reference LEFT [OUTER] JOIN table_reference USING (column_list)
table_reference NATURAL LEFT [OUTER] JOIN table_reference
{ oj table_reference LEFT OUTER JOIN table_reference ON conditional_expr }
```

The last `LEFT OUTER JOIN` syntax shown above exists only for compatibility with ODBC.

- A table reference may be aliased using `tbl_name AS alias_name` or `tbl_name alias_name`.

```
mysql> select t1.name, t2.salary from employee AS t1, info AS t2
        where t1.name = t2.name;
```

- `JOIN` and `,` (comma) are semantically equivalent. Both do a full join between the tables used. Normally, you specify how the tables should be linked in the `WHERE` condition.
- The `ON` conditional is any conditional of the form that may be used in a `WHERE` clause.
- If there is no matching record for the right table in a `LEFT JOIN`, a row with all columns set to `NULL` is used for the right table. You can use this fact to find records in a table that have no counterpart in another table:

```
mysql> select table1.* from table1
        LEFT JOIN table2 ON table1.id=table2.id
        where table2.id is NULL;
```

This example finds all rows in `table1` with an `id` value that is not present in `table2` (i.e., all rows in `table1` with no corresponding row in `table2`). This assumes that `table2.id` is declared `NOT NULL`, of course.

- The `USING column_list` clause names a list of columns that must exist in both tables. A `USING` clause such as:

```
A LEFT JOIN B USING (C1,C2,C3,...)
```

is defined to be semantically identical to an `ON` expression like this:

```
A.C1=B.C1 AND A.C2=B.C2 AND A.C3=B.C3,...
```

- The `NATURAL LEFT JOIN` of two tables is defined to be semantically equivalent to a `LEFT JOIN` with a `USING` clause that names all columns that exist in both tables.
- `STRAIGHT_JOIN` is identical to `JOIN`, except that the left table is always read before the right table. This can be used for those (few) cases where the join optimizer puts the tables in the wrong order.

Some examples:

```
mysql> select * from table1,table2 where table1.id=table2.id;
mysql> select * from table1 LEFT JOIN table2 ON table1.id=table2.id;
mysql> select * from table1 LEFT JOIN table2 USING (id);
mysql> select * from table1 LEFT JOIN table2 ON table1.id=table2.id
        LEFT JOIN table3 ON table2.id=table3.id;
```

See section [10.6 How MySQL optimizes LEFT JOIN](#).

## INSERT syntax

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
        [INTO] tbl_name [(col_name,...)]
        VALUES (expression,...),(...),...
or INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
        [INTO] tbl_name [(col_name,...)]
        SELECT ...
or INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
        [INTO] tbl_name
```

```
SET col_name=expression, col_name=expression, ...
```

INSERT inserts new rows into an existing table. The INSERT ... VALUES form of the statement inserts rows based on explicitly-specified values. The INSERT ... SELECT form inserts rows selected from another table or tables. The INSERT ... VALUES form with multiple value lists is supported in MySQL 3.22.5 or later. The col\_name=expression syntax is supported in MySQL 3.22.10 or later.

tbl\_name is the table into which rows should be inserted. The column name list or the SET clause indicates which columns the statement specifies values for.

- If you specify no column list for INSERT ... VALUES or INSERT ... SELECT, values for all columns must be provided in the VALUES() list or by the SELECT. If you don't know the order of the columns in the table, use DESCRIBE tbl\_name to find out.
- Any column not explicitly given a value is set to its default value. For example, if you specify a column list that doesn't name all the columns in the table, unnamed columns are set to their default values. Default value assignment is described in section CREATE TABLE syntax.
- An expression may refer to any column that was set earlier in a value list. For example, you can say this:

```
mysql> INSERT INTO tbl_name (col1,col2) VALUES(15,col1*2);
```

But not this:

```
mysql> INSERT INTO tbl_name (col1,col2) VALUES(col2*2,15);
```

- If you specify the keyword LOW\_PRIORITY, execution of the INSERT is delayed until no other clients are reading from the table.
- If you specify the keyword IGNORE in an INSERT with many value rows, any rows which duplicate an existing PRIMARY or UNIQUE key in the table are ignored and are not inserted. If you do not specify IGNORE, the insert is aborted if there is any row that duplicates an existing key value. You can check with the mysql\_info() how many rows were inserted into the table.
- If MySQL was configured using the DONT\_USE\_DEFAULT\_FIELDS option, INSERT statements generate an error unless you explicitly specify values for all columns that require a non-NULL value. See section 4.7.3 Typical configure options.
- The following conditions hold for a INSERT INTO ... SELECT statement:
  - The query cannot contain an ORDER BY clause.
  - The target table of the INSERT statement cannot appear in the FROM clause of the SELECT part of the query, because it's forbidden in ANSI SQL to SELECT from the same table into which you are INSERTING. (The problem is that the SELECT possibly would find records that were inserted earlier during the same run. When using sub-select clauses, the situation could easily be very confusing!)
  - AUTO\_INCREMENT columns work as usual.

If you use INSERT ... SELECT or a INSERT ... VALUES statement with multiple value lists, you can use the C API function mysql\_info() to get information about the query. The format of the information string is shown below:

```
Records: 100 Duplicates: 0 Warnings: 0
```

Duplicates indicates the number of rows that couldn't be inserted because they would duplicate some existing unique index value. Warnings indicates the number of attempts to insert column values that

were problematic in some way. Warnings can occur under any of the following conditions:

- Inserting `NULL` into a column that has been declared `NOT NULL`. The column is set to its default value.
- Setting a numeric column to a value that lies outside the column's range. The value is clipped to the appropriate endpoint of the range.
- Setting a numeric column to a value such as `'10.34 a'`. The trailing garbage is stripped and the remaining numeric part is inserted. If the value doesn't make sense as a number at all, the column is set to 0.
- Inserting a string into a `CHAR`, `VARCHAR`, `TEXT` or `BLOB` column that exceeds the column's maximum length. The value is truncated to the column's maximum length.
- Inserting a value into a date or time column that is illegal for the column type. The column is set to the appropriate "zero" value for the type.

The `DELAYED` option for the `INSERT` statement is a **MySQL**-specific option that is very useful if you have clients that can't wait for the `INSERT` to complete. This is common when you use **MySQL** for logging and you also periodically run `SELECT` statements that take a long time to complete. `DELAYED` was introduced in **MySQL** 3.22.15. It is a **MySQL** extension to ANSI SQL92.

Another major benefit of using `INSERT DELAYED` is that inserts from many clients are bundled together and written in one block. This is much faster than doing many separate inserts.

Note that currently the queued rows are only stored in memory until they are inserted into the table. This means that if you kill `mysqld` the hard way (`kill -9`) or if `mysqld` dies unexpectedly, any queued rows that weren't written to disk are lost!

The following happens when you use the `DELAYED` option to `INSERT` or `REPLACE`. In this description, the "thread" is the thread that received an `INSERT DELAYED` command and "handler" is the thread that handles all `INSERT DELAYED` statements for a particular table.

- When a thread executes a `DELAYED` statement for a table, a handler thread is created to handle all `DELAYED` statements for the table, if no such handler already exists.
- The thread checks whether or not the handler has acquired a `DELAYED` lock already; if not, it tells the handler thread to do so. The `DELAYED` lock can be obtained even if other threads have a `READ` or `WRITE` lock on the table. However, the handler will wait for all `ALTER TABLE` locks or `FLUSH TABLES` to ensure that the table structure is up to date.
- The thread executes the `INSERT` statement but instead of writing the row to the table it puts a copy of the final row into a queue that is managed by the handler thread. Any syntax errors are noticed by the thread and reported the client program.
- The client can't report the number of duplicates or the `AUTO_INCREMENT` value for the resulting row; it can't obtain them from the server, because the `INSERT` returns before the insert operation has been completed. If you use the C API, the `mysql_info()` function doesn't return anything meaningful, for the same reason.
- The update log is updated by the handler thread when the row is inserted into the table. In case of multiple-row inserts, the update log is updated when the first row is inserted.
- After every `delayed_insert_limit` rows are written, the handler checks whether or not any `SELECT` statements are still pending. If so, it allows these to execute before continuing.
- When the handler has no more rows in its queue, the table is unlocked. If no new `INSERT DELAYED` commands are received within `delayed_insert_timeout` seconds, the handler

terminates.

- If more than `delayed_queue_size` rows are pending already in a specific handler queue, the thread waits until there is room in the queue. This is useful to ensure that the `mysqld` server doesn't use all memory for the delayed memory queue.
- The handler thread will show up in the **MySQL** process list with `delayed_insert` in the `Command` column. It will be killed if you execute a `FLUSH TABLES` command or kill it with `KILL thread_id`. However, it will first store all queued rows into the table before exiting. During this time it will not accept any new `INSERT` commands from another thread. If you execute an `INSERT DELAYED` command after this, a new handler thread will be created.
- Note that the above means that `INSERT DELAYED` commands have higher priority than normal `INSERT` commands if there is an `INSERT DELAYED` handler already running! Other update commands will have to wait until the `INSERT DELAYED` queue is empty, someone kills the handler thread (with `KILL thread_id`) or someone executes `FLUSH TABLES`.
- The following status variables provide information about `INSERT DELAYED` commands:

<code>Delayed_insert_threads</code>	Number of handler threads
<code>Delayed_writes</code>	Number of rows written with <code>INSERT DELAYED</code>
<code>Not_flushed_delayed_rows</code>	Number of rows waiting to be written

You can view these variables by issuing a `SHOW STATUS` statement or by executing a `mysqladmin extended-status` command.

## REPLACE syntax

```
REPLACE [LOW_PRIORITY | DELAYED]
  [INTO] tbl_name [(col_name,...)]
  VALUES (expression,...)
or REPLACE [LOW_PRIORITY | DELAYED]
  [INTO] tbl_name [(col_name,...)]
  SELECT ...
or REPLACE [LOW_PRIORITY | DELAYED]
  [INTO] tbl_name
  SET col_name=expression, col_name=expression,...
```

`REPLACE` works exactly like `INSERT`, except that if an old record in the table has the same value as a new record on a unique index, the old record is deleted before the new record is inserted. See section [INSERT syntax](#).

## LOAD DATA INFILE syntax

```
LOAD DATA [LOCAL] INFILE 'file_name.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,...)]
```

The `LOAD DATA INFILE` statement reads rows from a text file into a table at a very high speed. If the `LOCAL` keyword is specified, the file is read from the client host. If `LOCAL` is not specified, the file must be located on the server. (`LOCAL` is available in **MySQL** 3.22.6 or later.)

For security reasons, when reading text files located on the server, the files must either reside in the database directory or be readable by all. Also, to use `LOAD DATA INFILE` on server files, you must have the **file** privilege on the server host. See section [6.6 How the privilege system works](#).

Using `LOCAL` will be a bit slower than letting the server access the files directly, since the contents of the file must travel from the client host to the server host. On the other hand, you do not need the **file** privilege to load local files.

You can also load data files by using the `mysqlimport` utility; it operates by sending a `LOAD DATA INFILE` command to the server. The `--local` option causes `mysqlimport` to read data files from the client host. You can specify the `--compress` option to get better performance over slow networks if the client and server support the compressed protocol.

When locating files on the server host, the server uses the following rules:

- If an absolute pathname is given, the server uses the pathname as is.
- If a relative pathname with one or more leading components is given, the server searches for the file relative to the server's data directory.
- If a filename with no leading components is given, the server looks for the file in the database directory of the current database.

Note that these rules mean a file given as `./myfile.txt` is read from the server's data directory, whereas a file given as `myfile.txt` is read from the database directory of the current database. Note also that for statements such as those below, the file is read from the database directory for `db1`, not `db2`:

```
mysql> USE db1;
mysql> LOAD DATA INFILE "./data.txt" INTO TABLE db2.my_table;
```

The `REPLACE` and `IGNORE` keywords control handling of input records that duplicate existing records on unique key values. If you specify `REPLACE`, new rows replace existing rows that have the same unique key value. If you specify `IGNORE`, input rows that duplicate an existing row on a unique key value are skipped. If you don't specify either option, an error occurs when a duplicate key value is found, and the rest of the text file is ignored.

If you load data from a local file using the `LOCAL` keyword, the server has no way to stop transmission of the file in the middle of the operation, so the default behavior is the same as if `IGNORE` is specified.

`LOAD DATA INFILE` is the complement of `SELECT ... INTO OUTFILE`. See section [SELECT syntax](#). To write data from a database to a file, use `SELECT ... INTO OUTFILE`. To read the file back into the database, use `LOAD DATA INFILE`. The syntax of the `FIELDS` and `LINES` clauses is the same for both commands. Both clauses are optional, but `FIELDS` must precede `LINES` if both are specified.

If you specify a `FIELDS` clause, each of its subclauses (`TERMINATED BY`, `[OPTIONALLY] ENCLOSED BY` and `ESCAPED BY`) is also optional, except that you must specify at least one of them.

If you don't specify a `FIELDS` clause, the defaults are the same as if you had written this:

```
FIELDS TERMINATED BY '\t' ENCLOSED BY " ESCAPED BY '\\'
```

If you don't specify a `LINES` clause, the default is the same as if you had written this:

LINES TERMINATED BY '\n'

In other words, the defaults cause `LOAD DATA INFILE` to act as follows when reading input:

- Look for line boundaries at newlines
- Break lines into fields at tabs
- Do not expect fields to be enclosed within any quoting characters
- Interpret occurrences of tab, newline or `\\` preceded by `\\` as literal characters that are part of field values

Conversely, the defaults cause `SELECT ... INTO OUTFILE` to act as follows when writing output:

- Write tabs between fields
- Do not enclose fields within any quoting characters
- Use `\\` to escape instances of tab, newline or `\\` that occur within field values
- Write newlines at the ends of lines

Note that to write `FIELDS ESCAPED BY '\\'`, you must specify two backslashes for the value to be read as a single backslash.

The `IGNORE number LINES` option can be used to ignore a header of column names at the start of the file:

```
mysql> LOAD DATA INFILE "/tmp/file_name" into table test IGNORE 1 LINES;
```

When you use `SELECT ... INTO OUTFILE` in tandem with `LOAD DATA INFILE` to write data from a database into a file and then read the file back into the database later, the field and line handling options for both commands must match. Otherwise, `LOAD DATA INFILE` will not interpret the contents of the file properly. Suppose you use `SELECT ... INTO OUTFILE` to write a file with fields delimited by commas:

```
mysql> SELECT * FROM table1 INTO OUTFILE 'data.txt'
        FIELDS TERMINATED BY ','
        FROM ...
```

To read the comma-delimited file back in, the correct statement would be:

```
mysql> LOAD DATA INFILE 'data.txt' INTO TABLE table2
        FIELDS TERMINATED BY ',';
```

If instead you tried to read in the file with the statement shown below, it wouldn't work because it instructs `LOAD DATA INFILE` to look for tabs between fields:

```
mysql> LOAD DATA INFILE 'data.txt' INTO TABLE table2
        FIELDS TERMINATED BY '\\t';
```

The likely result is that each input line would be interpreted as a single field.

`LOAD DATA INFILE` can be used to read files obtained from external sources, too. For example, a file in dBASE format will have fields separated by commas and enclosed in double quotes. If lines in the file are terminated by newlines, the command shown below illustrates the field and line handling options you would use to load the file:

```
mysql> LOAD DATA INFILE 'data.txt' INTO TABLE tbl_name
```

FIELDS TERMINATED BY ',' ENCLOSED BY ''  
LINES TERMINATED BY '\n';

Any of the field or line handling options may specify an empty string ("). If not empty, the FIELDS [OPTIONALLY] ENCLOSED BY and FIELDS ESCAPED BY values must be a single character. The FIELDS TERMINATED BY and LINES TERMINATED BY values may be more than one character. For example, to write lines that are terminated by carriage return-linefeed pairs, or to read a file containing such lines, specify a LINES TERMINATED BY '\r\n' clause.

FIELDS [OPTIONALLY] ENCLOSED BY controls quoting of fields. For output (SELECT ... INTO OUTFILE), if you omit the word OPTIONALLY, all fields are enclosed by the ENCLOSED BY character. An example of such output (using a comma as the field delimiter) is shown below:

```
"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"
```

If you specify OPTIONALLY, the ENCLOSED BY character is used only to enclose CHAR and VARCHAR fields:

```
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
```

Note that occurrences of the ENCLOSED BY character within a field value are escaped by prefixing them with the ESCAPED BY character. Also note that if you specify an empty ESCAPED BY value, it is possible to generate output that cannot be read properly by LOAD DATA INFILE. For example, the output just shown above would appear as shown below if the escape character is empty. Observe that the second field in the fourth line contains a comma following the quote, which (erroneously) appears to terminate the field:

```
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
```

For input, the ENCLOSED BY character, if present, is stripped from the ends of field values. (This is true whether or not OPTIONALLY is specified; OPTIONALLY has no effect on input interpretation.) Occurrences of the ENCLOSED BY character preceded by the ESCAPED BY character are interpreted as part of the current field value. In addition, duplicated ENCLOSED BY characters occurring within fields are interpreted as single ENCLOSED BY characters if the field itself starts with that character. For example, if ENCLOSED BY '' is specified, quotes are handled as shown below:

```
"The ""BIG"" boss" -> The "BIG" boss  
The "BIG" boss     -> The "BIG" boss  
The ""BIG"" boss   -> The ""BIG"" boss
```

FIELDS ESCAPED BY controls how to write or read special characters. If the FIELDS ESCAPED BY character is not empty, it is used to prefix the following characters on output:

- The FIELDS ESCAPED BY character
- The FIELDS [OPTIONALLY] ENCLOSED BY character
- The first character of the FIELDS TERMINATED BY and LINES TERMINATED BY values

- ASCII 0 (what is actually written following the escape character is ASCII '0', not a zero-valued byte)

If the `FIELDS ESCAPED BY` character is empty, no characters are escaped. It is probably not a good idea to specify an empty escape character, particularly if field values in your data contain any of the characters in the list just given.

For input, if the `FIELDS ESCAPED BY` character is not empty, occurrences of that character are stripped and the following character is taken literally as part of a field value. The exceptions are an escaped ``0`` or ``N`` (e.g., `\0` or `\N` if the escape character is ``\``). These sequences are interpreted as ASCII 0 (a zero-valued byte) and `NULL`. See below for the rules on `NULL` handling.

For more information about ``\``-escape syntax, see section [Literals: how to write strings and numbers](#).

In certain cases, field and line handling options interact:

- If `LINES TERMINATED BY` is an empty string and `FIELDS TERMINATED BY` is non-empty, lines are also terminated with `FIELDS TERMINATED BY`.
- If the `FIELDS TERMINATED BY` and `FIELDS ENCLOSED BY` values are both empty (`"`), a fixed-row (non-delimited) format is used. With fixed-row format, no delimiters are used between fields. Instead, column values are written and read using the "display" widths of the columns. For example, if a column is declared as `INT(7)`, values for the column are written using 7-character fields. On input, values for the column are obtained by reading 7 characters. Fixed-row format also affects handling of `NULL` values; see below.

Handling of `NULL` values varies, depending on the `FIELDS` and `LINES` options you use:

- For the default `FIELDS` and `LINES` values, `NULL` is written as `\N` for output and `\N` is read as `NULL` for input (assuming the `ESCAPED BY` character is ``\``).
- If `FIELDS ENCLOSED BY` is not empty, a field containing the literal word `NULL` as its value is read as a `NULL` value (this differs from the word `NULL` enclosed within `FIELDS ENCLOSED BY` characters, which is read as the string `'NULL'`).
- If `FIELDS ESCAPED BY` is empty, `NULL` is written as the word `NULL`.
- With fixed-row format (which happens when `FIELDS TERMINATED BY` and `FIELDS ENCLOSED BY` are both empty), `NULL` is written as an empty string. Note that this causes both `NULL` values and empty strings in the table to be indistinguishable when written to the file since they are both written as empty strings. If you need to be able to tell the two apart when reading the file back in, you should not use fixed-row format.

Some cases are not supported by `LOAD DATA INFILE`:

- Fixed-size rows (`FIELDS TERMINATED BY` and `FIELDS ENCLOSED BY` both empty) and `BLOB` or `TEXT` columns.
- If you specify one separator that is the same as or a prefix of another, `LOAD DATA INFILE` won't be able to interpret the input properly. For example, the following `FIELDS` clause would cause problems:

```
FIELDS TERMINATED BY ''' ENCLOSED BY '''
```

- If `FIELDS ESCAPED BY` is empty, a field value that contains an occurrence of `FIELDS ENCLOSED`

BY OR LINES TERMINATED BY followed by the FIELDS TERMINATED BY value will cause LOAD DATA INFILE to stop reading a field or line too early. This happens because LOAD DATA INFILE cannot properly determine where the field or line value ends.

The following example loads all columns of the `persondata` table:

```
mysql> LOAD DATA INFILE 'persondata.txt' INTO TABLE persondata;
```

No field list is specified, so LOAD DATA INFILE expects input rows to contain a field for each table column. The default FIELDS and LINES values are used.

If you wish to load only some of a table's columns, specify a field list:

```
mysql> LOAD DATA INFILE 'persondata.txt'  
      INTO TABLE persondata (col1,col2,...);
```

You must also specify a field list if the order of the fields in the input file differs from the order of the columns in the table. Otherwise, **MySQL** cannot tell how to match up input fields with table columns.

If a row has too few fields, the columns for which no input field is present are set to default values. Default value assignment is described in section [CREATE TABLE syntax](#).

An empty field value is interpreted differently than if the field value is missing:

- For string types, the column is set to the empty string.
- For numeric types, the column is set to 0.
- For date and time types, the column is set to the appropriate "zero" value for the type. See section [Date and time types](#).

TIMESTAMP columns are only set to the current date and time if there is a NULL value for the column, or (for the first TIMESTAMP column only) if the TIMESTAMP column is left out from the field list when a field list is specified.

If an input row has too many fields, the extra fields are ignored and the number of warnings is incremented.

LOAD DATA INFILE regards all input as strings, so you can't use numeric values for ENUM or SET columns the way you can with INSERT statements. All ENUM and SET values must be specified as strings!

If you are using the C API, you can get information about the query by calling the API function `mysql_info()` when the LOAD DATA INFILE query finishes. The format of the information string is shown below:

```
Records: 1 Deleted: 0 Skipped: 0 Warnings: 0
```

Warnings occur under the same circumstances as when values are inserted via the INSERT statement (see section [INSERT syntax](#)), except that LOAD DATA INFILE also generates warnings when there are too few or too many fields in the input row. The warnings are not stored anywhere; the number of warnings can only be used as an indication if everything went well. If you get warnings and want to know exactly why you got them, one way to do this is to use `SELECT ... INTO OUTFILE` into another file and compare this to your original input file.

For more information about the efficiency of `INSERT` versus `LOAD DATA INFILE` and speeding up `LOAD DATA INFILE`, see section [10.11 How to arrange a table to be as fast/small as possible](#).

## UPDATE syntax

```
UPDATE [LOW_PRIORITY] tbl_name SET col_name1=expr1,col_name2=expr2,...
    [WHERE where_definition]
```

`UPDATE` updates columns in existing table rows with new values. The `SET` clause indicates which columns to modify and the values they should be given. The `WHERE` clause, if given, specifies which rows should be updated. Otherwise all rows are updated.

If you specify the keyword `LOW_PRIORITY`, execution of the `UPDATE` is delayed until no other clients are reading from the table.

If you access a column from `tbl_name` in an expression, `UPDATE` uses the current value of the column. For example, the following statement sets the `age` column to one more than its current value:

```
mysql> UPDATE persondata SET age=age+1;
```

`UPDATE` assignments are evaluated from left to right. For example, the following statement doubles the `age` column, then increments it:

```
mysql> UPDATE persondata SET age=age*2, age=age+1;
```

If you set a column to the value it currently has, **MySQL** notices this and doesn't update it.

`UPDATE` returns the number of rows that were actually changed. In **MySQL** 3.22 or later, the C API function `mysql_info()` returns the number of rows that were matched and updated and the number of warnings that occurred during the `UPDATE`.

## USE syntax

```
USE db_name
```

The `USE db_name` statement tells **MySQL** to use the `db_name` database as the default database for subsequent queries. The database remains current until the end of the session, or until another `USE` statement is issued:

```
mysql> USE db1;
mysql> SELECT count(*) FROM mytable;      # selects from db1.mytable
mysql> USE db2;
mysql> SELECT count(*) FROM mytable;      # selects from db2.mytable
```

Making a particular database current by means of the `USE` statement does not preclude you from accessing tables in other databases. The example below accesses the `author` table from the `db1` database and the `editor` table from the `db2` database:

```
mysql> USE db1;
mysql> SELECT author_name,editor_name FROM author,db2.editor
    WHERE author.editor_id = db2.editor.editor_id;
```

The USE statement is provided for Sybase compatibility.

## FLUSH syntax (clearing caches)

```
FLUSH flush_option [,flush_option]
```

You should use the FLUSH command if you want to clear some of the internal caches MySQL uses. To execute FLUSH, you must have the **reload** privilege.

flush\_option can be any of the following:

HOSTS	Empties the host cache tables. You should flush the host tables if some of your hosts change IP number or if you get the error message <code>Host ... is blocked</code> . When more than <code>max_connect_errors</code> errors occur in a row for a given host while connection to the MySQL server, MySQL assumes something is wrong and blocks the host from further connection requests. Flushing the host tables allows the host to attempt to connect again. See section 17.2.3 <code>Host '...' is blocked error</code> .) You can start <code>mysqld</code> with <code>-O max_connection_errors=999999999</code> to avoid this error message.
LOGS	Closes and reopens the standard and update log files. If you have specified the update log file without an extension, the extension number of the new update log file will be incremented by one relative to the previous file.
PRIVILEGES	Reloads the privileges from the grant tables in the <code>mysql</code> database.
TABLES	Closes all open tables.
STATUS	Resets most status variables to zero.

You can also access each of the commands shown above with the `mysqladmin` utility, using the `flush-hosts`, `flush-logs`, `reload` or `flush-tables` commands.

## KILL syntax

```
KILL thread_id
```

Each connection to `mysqld` runs in a separate thread. You can see which threads are running with the `SHOW PROCESSLIST` command, and kill a thread with the `KILL thread_id` command.

If you have the **process** privilege, you can see and kill all threads. Otherwise, you can see and kill only your own threads.

You can also use the `mysqladmin processlist` and `mysqladmin kill` commands to examine and kill threads.

## SHOW syntax (Get information about tables, columns,...)

```
SHOW DATABASES [LIKE wild]
or SHOW TABLES [FROM db_name] [LIKE wild]
or SHOW COLUMNS FROM tbl_name [FROM db_name] [LIKE wild]
or SHOW INDEX FROM tbl_name [FROM db_name]
```

```
or SHOW STATUS
or SHOW VARIABLES [LIKE wild]
or SHOW PROCESSLIST
or SHOW TABLE STATUS [FROM db_name] [LIKE wild]
```

SHOW provides information about databases, tables, columns or the server. If the `LIKE wild` part is used, the `wild` string can be a string that uses the SQL ``%`` and ``_`` wildcard characters.

You can use `db_name.tbl_name` as an alternative to the `tbl_name FROM db_name` syntax. These two statements are equivalent:

```
mysql> SHOW INDEX FROM mytable FROM mydb;
mysql> SHOW INDEX FROM mydb.mytable;
```

SHOW DATABASES lists the databases on the **MySQL** server host. You can also get this list using the `mysqlshow` command.

SHOW TABLES lists the tables in a given database. You can also get this list using the `mysqlshow db_name` command.

**Note:** If a user doesn't have any privileges for a table, the table will not show up in the output from `SHOW TABLES` or `mysqlshow db_name`.

SHOW COLUMNS lists the columns in a given table. If the column types are different than you expect them to be based on a `CREATE TABLE` statement, note that MySQL sometimes changes column types. See section [Silent column specification changes](#).

The `DESCRIBE` statement provides information similar to `SHOW COLUMNS`. See section [DESCRIBE syntax \(Get information about columns\)](#).

SHOW TABLE STATUS (new in 3.23) works like `SHOW STATUS`, but provides a lot of information about each table. You can also get this list using the `mysqlshow --status db_name` command. The following columns are returned:

Name	Name of the table
Type	Type of table (NISAM, MYISAM or HEAP)
Rows	Number of rows
Avg_row_length	Average row length
Data_length	Length of the data file
Max_data_length	Max length of the data file
Index_length	Length of the index file
Data_free	Number of allocated but not used bytes
Auto_increment	Next autoincrement value
Create_time	When the table was created
Update_time	When the data file was last updated
Check_time	When one last run a check on the table
Create_min_rows	The "min_rows" option used when creating the table
Create_max_rows	The "max_rows" option used when creating the table
Create_avg_row_length	The "avg_row_length" option used when creating the table
Comment	The comment used when creating the table (or some information why MySQL couldn't access the table information).

SHOW FIELDS is a synonym for SHOW COLUMNS and SHOW KEYS is a synonym for SHOW INDEX. You can also list a table's columns or indexes with `mysqlshow db_name tbl_name` or `mysqlshow -k db_name tbl_name`.

SHOW INDEX returns the index information in a format that closely resembles the SQLStatistics call in ODBC. The following columns are returned:

Table	Name of the table
Non_unique	0 if the index can't contain duplicates.
Key_name	Name of the index
Seq_in_index	Column sequence number in index, starting with 1.
Column_name	Column name.
Collation	How the column is sorted in the index. In <b>MySQL</b> , this can have values A (Ascending) or NULL (Not sorted).
Cardinality	Number of unique values in the index. This is updated by running <code>isamchk -a</code> .
Sub_part	Number of indexed characters if the column is only partly indexed. NULL if the entire key is indexed.

SHOW STATUS provides server status information (like `mysqladmin extended-status`). The output resembles that shown below, though the format and numbers may differ somewhat:

```
+-----+-----+
| Variable_name          | Value |
+-----+-----+
```

Aborted_clients	0
Aborted_connects	0
Created_tmp_tables	0
Delayed_insert_threads	0
Delayed_writes	0
Delayed_errors	0
Flush_commands	2
Handler_delete	2
Handler_read_first	0
Handler_read_key	1
Handler_read_next	0
Handler_read_rnd	35
Handler_update	0
Handler_write	2
Key_blocks_used	0
Key_read_requests	0
Key_reads	0
Key_write_requests	0
Key_writes	0
Max_used_connections	1
Not_flushed_key_blocks	0
Not_flushed_delayed_rows	0
Open_tables	1
Open_files	2
Open_streams	0
Opened_tables	11
Questions	14
Running_threads	1
Slow_queries	0
Uptime	149111

The status variables listed above have the following meaning:

Aborted_clients	Number of connections that has been aborted because the client has died without closing the connection properly.
Aborted_connects	Number of tries to connect to the MySQL server that has failed.
Created_tmp_tables	Number of implicit temporary tables that has been created while executing statements.
Delayed_insert_threads	Number of delayed insert handler threads in use.
Delayed_writes	Number of rows written with INSERT DELAYED.
Delayed_errors	Number of rows written with INSERT DELAYED for which some error occurred (probably duplicate key).
Flush_commands	Number of executed FLUSH commands.
Handler_delete	Number of requests to delete a row from a table.
Handler_read_first	Number of request to read first the row in a table.
Handler_read_key	Number of request to read a row based on a key.
Handler_read_next	Number of request to read next row in key order.
Handler_read_rnd	Number of request to read a row based on a fixed position.
Handler_update	Number of requests to update a row in a table.
Handler_write	Number of requests to insert a row in a table.
Key_blocks_used	The number of used blocks in the key cache.
Key_read_requests	The number of request to read a key block from the cache.
Key_reads	The number of physical reads of a key block from disk.
Key_write_requests	The number of request to write a key block to the cache.
Key_writes	The number of physical writes of a key block to disk.
Max_used_connections	The maximum number of connections that has been in use simultaneously.
Not_flushed_key_blocks	Keys blocks in the key cache that has changed but hasn't yet been flushed to disk.
Not_flushed_delayed_rows	Number of rows waiting to be written in INSERT DELAY queues.
Open_tables	Number of tables that are open.
Open_files	Number of files that are open.
Open_streams	Number of streams that are open (used mainly for logging)
Opened_tables	Number of tables that has been opened.
Questions	Number of questions asked from to the server.
Running_threads	Number of currently open connections.
Slow_queries	Number of queries that has taken more than long_query_time
Uptime	How many seconds the server has been up.

Some comments about the above:

- If `Opened_tables` is big, then your `table_cache` variable is probably too small.

- If `key_reads` is big, then your `key_cache` is probably too small. The cache hit rate can be calculated with `key_reads/key_read_requests`.
- If `Handler_read_rnd` is big, then you have a probably a lot of queries that requires MySQL to scan whole tables or you have joins that doesn't use keys properly.

`SHOW VARIABLES` shows the values of the some of **MySQL** system variables. You can also get this information using the `mysqladmin variables` command. If the default values are unsuitable, you can set most of these variables using command-line options when `mysqld` starts up. The output resembles that shown below, though the format and numbers may differ somewhat:

Variable_name	Value
back_log	5
connect_timeout	5
basedir	/my/monty/
datadir	/my/monty/data/
delayed_insert_limit	100
delayed_insert_timeout	300
delayed_queue_size	1000
join_buffer_size	131072
flush_time	0
key_buffer_size	1048540
language	/my/monty/share/english/
log	OFF
log_update	OFF
long_query_time	10
low_priority_updates	OFF
max_allowed_packet	1048576
max_connections	100
max_connect_errors	10
max_delayed_threads	20
max_heap_table_size	16777216
max_join_size	4294967295
max_sort_length	1024
max_tmp_tables	32
net_buffer_length	16384
port	3306
protocol-version	10
record_buffer	131072
skip_locking	ON
socket	/tmp/mysql.sock
sort_buffer	2097116
table_cache	64
thread_stack	131072
tmp_table_size	1048576
tmpdir	/machine/tmp/
version	3.23.0-alpha-debug
wait_timeout	28800

See section [10.1 Tuning server parameters](#).

`SHOW PROCESSLIST` shows you which threads are running. You can also get this information using the `mysqladmin processlist` command. If you have the **process** privilege, you can see all threads. Otherwise, you can see only your own threads. See section [KILL syntax](#).

## **EXPLAIN syntax (Get information about a SELECT)**

```
EXPLAIN SELECT select_options
```

When you precede a `SELECT` statement with the keyword `EXPLAIN`, **MySQL** explains how it would process the `SELECT`, providing information about how tables are joined and in which order.

With the help of `EXPLAIN`, you can see when you must add indexes to tables to get a faster `SELECT` that uses indexes to find the records. You can also see if the optimizer joins the tables in an optimal order. To force the optimizer to use a specific join order for a `SELECT` statement, add a `STRAIGHT_JOIN` clause.

For non-simple joins, `EXPLAIN` returns a row of information for each table used in the `SELECT` statement. The tables are listed in the order they would be read. **MySQL** resolves all joins using a single-sweep multi-join method. This means that **MySQL** reads a row from the first table, then finds a matching row in the second table, then in the third table and so on. When all tables are processed, it outputs the selected columns and backtracks through the table list until a table is found for which there are more matching rows. The next row is read from this table and the process continues with the next table.

Output from `EXPLAIN` includes the following columns:

`table`

The table to which the row of output refers.

`type` The join type. Information about the various types is given below.

`possible_keys`

The `possible_keys` column indicates which indexes **MySQL** could use to find the rows in the table. If this column is empty, there are no relevant indexes. In this case, you may be able to improve the performance of your query by examining the `WHERE` clause to see if it refers to some column or columns that would be suitable for indexing. If so, create an appropriate index and check the query with `EXPLAIN` again. To see what indexes a table has, use `SHOW INDEX FROM tbl_name`.

`key` The `key` column indicates the key that **MySQL** actually decided to use. The key is `NULL` if no index was chosen.

`key_len`

The `key_len` column indicates the length of the key that **MySQL** decided to use. The length is `NULL` if the key is `NULL`.

`ref` The `ref` column shows which columns or constants are used with the `key` to select rows from the table.

`rows` The `rows` column indicates the number of rows **MySQL** must examine to execute the query.

`Extra`

If the `Extra` column includes the text `Only index`, this means that information is retrieved from the table using only information in the index tree. Normally, this is much faster than scanning the entire table. If the `Extra` column includes the text `where used`, it means that a `WHERE` clause will be used to restrict which rows will be matched against the next table or sent to the client.

The different join types are listed below, ordered from best to worst type:

`system`

The table has only one row (= system table). This is a special case of the `const` join type.

`const`

The table has at most one matching row, which will be read at the start of the query. Since there is only one row, values from the column in this row can be regarded as constants by the rest of the

optimizer. `const` tables are very fast as they are read only once!

`eq_ref`

One row will be read from this table for each combination of rows from the previous tables. This is the best possible join type, other than the `const` types. It is used when all parts of an index are used by the join and the index is `UNIQUE` or a `PRIMARY KEY`.

`ref`

All rows with matching index values will be read from this table for each combination of rows from the previous tables. `ref` is used if the join uses only a leftmost prefix of the key, or if the key is not `UNIQUE` or a `PRIMARY KEY` (in other words, if the join cannot select a single row based on the key value). If the key that is used matches only a few rows, this join type is good.

`range`

Only rows that are in a given range will be retrieved, using an index to select the rows. The `ref` column indicates which index is used.

`index`

This is the same as `ALL`, except that only the index tree is scanned. This is usually faster than `ALL`, as the index file is usually smaller than the data file.

`ALL`

A full table scan will be done for each combination of rows from the previous tables. This is normally not good if the table is the first table not marked `const`, and usually **very** bad in all other cases. You normally can avoid `ALL` by adding more indexes, so that the row can be retrieved based on constant values or column values from earlier tables.

You can get a good indication of how good a join is by multiplying all values in the `rows` column of the `EXPLAIN` output. This should tell you roughly how many rows **MySQL** must examine to execute the query. This number is also used when you restrict queries with the `max_join_size` variable. See section [10.1 Tuning server parameters](#).

The following example shows how a `JOIN` can be optimized progressively using the information provided by `EXPLAIN`.

Suppose you have the `SELECT` statement shown below, that you examine using `EXPLAIN`:

```
EXPLAIN SELECT tt.TicketNumber, tt.TimeIn,
             tt.ProjectReference, tt.EstimatedShipDate,
             tt.ActualShipDate, tt.ClientID,
             tt.ServiceCodes, tt.RepetitiveID,
             tt.CurrentProcess, tt.CurrentDPPerson,
             tt.RecordVolume, tt.DPPrinted, et.COUNTRY,
             et_1.COUNTRY, do.CUSTNAME
FROM tt, et, et AS et_1, do
WHERE tt.SubmitTime IS NULL
      AND tt.ActualPC = et.EMPLOYID
      AND tt.AssignedPC = et_1.EMPLOYID
      AND tt.ClientID = do.CUSTNMBR;
```

For this example, assume that:

- The columns being compared have been declared as follows:

Table	Column	Column type
tt	ActualPC	CHAR(10)
tt	AssignedPC	CHAR(10)
tt	ClientID	CHAR(10)
et	EMPLOYID	CHAR(15)
do	CUSTNMBR	CHAR(15)

- The tables have the indexes shown below:

Table	Index
tt	ActualPC
tt	AssignedPC
tt	ClientID
et	EMPLOYID (primary key)
do	CUSTNMBR (primary key)

- The `tt.ActualPC` values aren't evenly distributed.

Initially, before any optimizations have been performed, the `EXPLAIN` statement produces the following information:

```

table type possible_keys          key  key_len ref  rows  Extra
et     ALL  PRIMARY                NULL NULL  NULL  74
do     ALL  PRIMARY                NULL NULL  NULL  2135
et_1   ALL  PRIMARY                NULL NULL  NULL  74
tt     ALL  AssignedPC,ClientID,ActualPC  NULL NULL  NULL  3872
      range checked for each record (key map: 35)

```

Since `type` is `ALL` for each table, this output indicates that **MySQL** is doing a full join for all tables! This will take quite a long time, as the product of the number of rows in each table must be examined! For the case at hand, this is  $74 * 2135 * 74 * 3872 = 45,268,558,720$  rows. If the tables were bigger, you can only imagine how long it would take...

One problem here is that **MySQL** can't (yet) use indexes on columns efficiently if they are declared differently. In this context, `VARCHAR` and `CHAR` are the same unless they are declared as different lengths. Since `tt.ActualPC` is declared as `CHAR(10)` and `et.EMPLOYID` is declared as `CHAR(15)`, there is a length mismatch.

To fix this disparity between column lengths, use `ALTER TABLE` to lengthen `ActualPC` from 10 characters to 15 characters:

```
mysql> ALTER TABLE tt MODIFY ActualPC VARCHAR(15);
```

Now `tt.ActualPC` and `et.EMPLOYID` are both `VARCHAR(15)`. Executing the `EXPLAIN` statement again produces this result:

```

table type possible_keys          key  key_len ref  rows  Extra
tt     ALL  AssignedPC,ClientID,ActualPC  NULL NULL  NULL  3872  where used
do     ALL  PRIMARY                NULL NULL  NULL  2135
      range checked for each record (key map: 1)
et_1   ALL  PRIMARY                NULL NULL  NULL  74

```

```

range checked for each record (key map: 1)
et      eq_ref PRIMARY          PRIMARY 15          tt.ActualPC 1

```

This is not perfect, but is much better (the product of the `rows` values is now less by a factor of 74). This version is executed in a couple of seconds.

A second alteration can be made to eliminate the column length mismatches for the `tt.AssignedPC = et_1.EMPLOYID` and `tt.ClientID = do.CUSTNMBR` comparisons:

```

mysql> ALTER TABLE tt MODIFY AssignedPC VARCHAR(15),
        MODIFY ClientID   VARCHAR(15);

```

Now `EXPLAIN` produces the output shown below:

```

table type  possible_keys  key      key_len ref          rows  Extra
et     ALL     PRIMARY       NULL     NULL  NULL        74
tt     ref     AssignedPC,ClientID,ActualPC ActualPC 15  et.EMPLOYID 52  where used
et_1   eq_ref  PRIMARY       PRIMARY 15      tt.AssignedPC 1
do     eq_ref  PRIMARY       PRIMARY 15      tt.ClientID   1

```

This is "almost" as good as it can get.

The remaining problem is that, by default, **MySQL** assumes that values in the `tt.ActualPC` column are evenly distributed, and that isn't the case for the `tt` table. Fortunately, it is easy to tell **MySQL** about this:

```

shell> isamchk --analyze PATH_TO_MYSQL_DATABASE/tt
shell> mysqladmin refresh

```

Now the join is "perfect", and `EXPLAIN` produces this result:

```

table type  possible_keys  key      key_len ref          rows  Extra
tt     ALL     AssignedPC,ClientID,ActualPC NULL  NULL  NULL        3872  where used
et     eq_ref  PRIMARY       PRIMARY 15      tt.ActualPC 1
et_1   eq_ref  PRIMARY       PRIMARY 15      tt.AssignedPC 1
do     eq_ref  PRIMARY       PRIMARY 15      tt.ClientID   1

```

Note that the `rows` column in the output from `EXPLAIN` is an "educated guess" from the **MySQL** join optimizer; To optimize a query, you should check if the numbers are even close to the truth. If not, you may get better performance by using `STRAIGHT_JOIN` in your `SELECT` statement and trying to list the tables in a different order in the `FROM` clause.

## DESCRIBE syntax (Get information about columns)

```
{DESCRIBE | DESC} tbl_name {col_name | wild}
```

`DESCRIBE` provides information about a table's columns. `col_name` may be a column name or a string containing the SQL ``%'` and ``_'` wildcard characters.

If the column types are different than you expect them to be based on a `CREATE TABLE` statement, note that **MySQL** sometimes changes column types. See section [Silent column specification changes](#).

This statement is provided for Oracle compatibility.

The `SHOW` statement provides similar information. See section [SHOW syntax \(Get information about tables, columns,...\)](#).

## LOCK TABLES/UNLOCK TABLES syntax

```
LOCK TABLES tbl_name [AS alias] {READ | [LOW_PRIORITY] WRITE}
            [, tbl_name {READ | [LOW_PRIORITY] WRITE} ...]
...
UNLOCK TABLES
```

`LOCK TABLES` locks tables for the current thread. `UNLOCK TABLES` releases any locks held by the current thread. All tables that are locked by the current thread are automatically unlocked when the thread issues another `LOCK TABLES`, or when the connection to the server is closed.

If a thread obtains a `READ` lock on a table, that thread (and all other threads) can only read from the table. If a thread obtains a `WRITE` lock on a table, then only the thread holding the lock can `READ` from or `WRITE` to the table. Other threads are blocked.

Each thread waits (without timing out) until it obtains all the locks it has requested.

`WRITE` locks normally have higher priority than `READ` locks, to ensure that updates are processed as soon as possible. This means that if one thread obtains a `READ` lock and then another thread requests a `WRITE` lock, subsequent `READ` lock requests will wait until the `WRITE` thread has gotten the lock and released it. You can use `LOW_PRIORITY WRITE` locks to allow other threads to obtain `READ` locks while the thread is waiting for the `WRITE` lock. You should only use `LOW_PRIORITY WRITE` locks if you are sure that there will eventually be a time when no threads will have a `READ` lock.

When you use `LOCK TABLES`, you must lock all tables that you are going to use! If you are using a table multiple times in a query (with aliases), you must get a lock for each alias! This policy ensures that table locking is deadlock free.

Note that you should **NOT** lock any tables that you are using with `INSERT DELAYED`. This is because that in this case the `INSERT` is done by a separate thread.

Normally, you don't have to lock tables, as all single `UPDATE` statements are atomic; no other thread can interfere with any other currently executing `SQL` statement. There are a few cases when you would like to lock tables anyway:

- If you are going to run many operations on a bunch of tables, it's much faster to lock the tables you are going to use. The downside is, of course, that no other thread can update a `READ`-locked table and no other thread can read a `WRITE`-locked table.
- **MySQL** doesn't support a transaction environment, so you must use `LOCK TABLES` if you want to ensure that no other thread comes between a `SELECT` and an `UPDATE`. The example shown below requires `LOCK TABLES` in order to execute safely:

```
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;
```

Without `LOCK TABLES`, there is a chance that another thread might insert a new row in the `trans` table between execution of the `SELECT` and `UPDATE` statements.

By using incremental updates (`UPDATE customer SET value=value+new_value`) or the `LAST_INSERT_ID()` function, you can avoid using `LOCK TABLES` in many cases.

You can also solve some cases by using the user-level lock functions `GET_LOCK()` and `RELEASE_LOCK()`. These locks are saved in a hash table in the server and implemented with `pthread_mutex_lock()` and `pthread_mutex_unlock()` for high speed. See section [Miscellaneous functions](#).

See section [10.10 How MySQL locks tables](#), for more information on locking policy.

## SET OPTION syntax

```
SET [OPTION] SQL_VALUE_OPTION= value, ...
```

`SET OPTION` sets various options that affect the operation of the server or your client. Any option you set remains in effect until the current session ends, or until you set the option to a different value.

```
CHARACTER SET character_set_name | DEFAULT
```

This maps all strings from and to the client with the given mapping. Currently the only option for `character_set_name` is `cp1251_koi8`, but you can easily add new mappings by editing the ``sql/convert.cc'` file in the **MySQL** source distribution. The default mapping can be restored by using a `character_set_name` value of `DEFAULT`. Note that the syntax for setting the `CHARACTER SET` option differs from the syntax for setting the other options.

```
PASSWORD = PASSWORD('some password')
```

Set the password for the current user. Any non-anonymous user can change his own password!

```
PASSWORD FOR user = PASSWORD('some password')
```

Set the password for a specific user on the current server host. Only a user with access to the `mysql` database can do this. The user should be given in `user@hostname` format, where `user` and `hostname` are exactly as they are listed in the `User` and `Host` columns of the `mysql.user` table entry. For example, if you had an entry with `User` and `Host` fields of `'bob'` and `'%.loc.gov'`, you would write:

```
mysql> SET PASSWORD FOR bob@%.loc.gov = PASSWORD("newpass");
```

```
SQL_BIG_TABLES = 0 | 1
```

If set to 1, all temporary tables are stored on disk rather than in memory. This will be a little slower, but you will not get the error `The table tbl_name is full for big SELECT operations that require a large temporary table`. The default value for a new connection is 0 (i.e., use in-memory temporary tables).

```
SQL_BIG_SELECTS = 0 | 1
```

If set to 1, **MySQL** will abort if a `SELECT` is attempted that probably will take a very long time. This is useful when an inadvisable `WHERE` statement has been issued. A big query is defined as a `SELECT` that probably will have to examine more than `max_join_size` rows. The default value for a new connection is 0 (which will allow all `SELECT` statements).

```
SQL_LOW_PRIORITY_UPDATES = 0 | 1
```

If set to 1, all `INSERT`, `UPDATE` and `DELETE` statements wait until there is no pending `SELECT` on the affected table.

```
SQL_SELECT_LIMIT = value | DEFAULT
```

The maximum number of records to return from `SELECT` statements. If a `SELECT` has a `LIMIT` clause, the `LIMIT` takes precedence over the value of `SQL_SELECT_LIMIT`. The default value for a new connection is "unlimited". If you have changed the limit, the default value can be restored by using a `SQL_SELECT_LIMIT` value of `DEFAULT`.

```
SQL_LOG_OFF = 0 | 1
```

If set to 1, no logging will be done to the standard log for this client, if the client has the **process** privilege. This does not affect the update log!

```
SQL_LOG_UPDATE = 0 | 1
```

If set to 0, no logging will be done to the update log for the client, if the client has the **process** privilege. This does not affect the standard log!

```
TIMESTAMP = timestamp_value | DEFAULT
```

Set the time for this client. This is used to get the original timestamp if you use the update log to restore rows.

```
LAST_INSERT_ID = #
```

Set the value to be returned from `LAST_INSERT_ID()`. This is stored in the update log when you use `LAST_INSERT_ID()` in a command that updates a table.

```
INSERT_ID = #
```

Set the value to be used by the following `INSERT` command when inserting an `AUTO_INCREMENT` value. This is mainly used with the update log.

## GRANT and REVOKE syntax

```
GRANT priv_type [(column_list)] [, priv_type [(column_list)] ...]
  ON {tbl_name | * | *.* | db_name.*}
  TO user_name [IDENTIFIED BY 'password']
     [, user_name [IDENTIFIED BY 'password'] ...]
  [WITH GRANT OPTION]
```

```
REVOKE priv_type [(column_list)] [, priv_type [(column_list)] ...]
  ON {tbl_name | * | *.* | db_name.*}
  FROM user_name [, user_name ...]
```

`GRANT` is implemented in **MySQL** 3.22.11 or later. For earlier **MySQL** versions, the `GRANT` statement does nothing.

The `GRANT` and `REVOKE` commands allow system administrators to grant and revoke rights to **MySQL** users at four privilege levels:

### Global level

Global privileges apply to all databases on a given server. These privileges are stored in the `mysql.user` table.

### Database level

Database privileges apply to all tables in a given database. These privileges are stored in the `mysql.db` and `mysql.host` tables.

### Table level

Table privileges apply to all columns in a given table. These privileges are stored in the `mysql.tables_priv` table.

### Column level

Column privileges apply to single columns in a given table. These privileges are stored in the `mysql.columns_priv` table.

For examples of how GRANT works, see section [6.11 Adding new user privileges to MySQL](#).

For the GRANT and REVOKE statements, `priv_type` may be specified as any of the following:

ALL PRIVILEGES	FILE	RELOAD
ALTER	INDEX	SELECT
CREATE	INSERT	SHUTDOWN
DELETE	PROCESS	UPDATE
DROP	REFERENCES	USAGE

ALL is a synonym for ALL PRIVILEGES. REFERENCES is not yet implemented. USAGE is currently a synonym for "no privileges". It can be used when you want to create a user that has no privileges.

To revoke the **grant** privilege from a user, use a `priv_type` value of GRANT OPTION:

```
REVOKE GRANT OPTION ON ... FROM ...;
```

The only `priv_type` values you can specify for a table are SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, GRANT, INDEX and ALTER.

The only `priv_type` values you can specify for a column (that is, when you use a `column_list` clause) are SELECT, INSERT and UPDATE.

You can set global privileges by using `ON *.*` syntax. You can set database privileges by using `ON db_name.*` syntax. If you specify `ON *` and you have a current database, you will set the privileges for that database. (**Warning:** If you specify `ON *` and you *don't* have a current database, you will affect the global privileges!)

In order to accommodate granting rights to users from arbitrary hosts, **MySQL** supports specifying the `user_name` value in the form `user@host`. If you want to specify a user string containing special characters (such as ``-``), or a host string containing special characters or wildcard characters (such as ``%``), you can quote the user or host name (e.g., `'test-user'@'test-hostname'`).

You can specify wildcards in the hostname. For example, `user@%.loc.gov` applies to `user` for any host in the `loc.gov` domain, and `user@144.155.166.%` applies to `user` for any host in the `144.155.166` class C subnet.

The simple form `user` is a synonym for `user@%``. **Note:** If you allow anonymous users to connect to the **MySQL** server (which is the default), you should also add all local users as `user@localhost` because otherwise the anonymous user entry for the local host in the `mysql.user` table will be used when the user tries to log into the **MySQL** server from the local machine! Anonymous users are defined by inserting entries with `User=""` into the `mysql.user` table. You can verify if this applies to you by executing this query:

```
mysql> SELECT Host,User FROM mysql.user WHERE User="";
```

For the moment, GRANT only supports host, table, database and column names up to 60 characters long. A user name can be up to 16 characters.

The privileges for a table or column are formed from the logical OR of the privileges at each of the four privilege levels. For example, if the `mysql.user` table specifies that a user has a global **select** privilege, this can't be denied by an entry at the database, table or column level.

The privileges for a column can be calculated as follows:

```
global privileges
OR (database privileges AND host privileges)
OR table privileges
OR column privileges
```

In most cases, you grant rights to a user at only one of the privilege levels, so life isn't normally as complicated as above. :) The details of the privilege-checking procedure are presented in section [6 The MySQL access privilege system](#).

If you grant privileges for a user/hostname combination that does not exist in the `mysql.user` table, an entry is added and remains there until deleted with a `DELETE` command. In other words, `GRANT` may create `user` table entries, but `REVOKE` will not remove them; you must do that explicitly using `DELETE`.

In **MySQL** 3.22.12 or later, if a new user is created or if you have global grant privileges, the user's password will be set to the password specified by the `IDENTIFIED BY` clause, if one is given. If the user already had a password, it is replaced by the new one.

**Warning:** If you create a new user but do not specify an `IDENTIFIED BY` clause, the user has no password. This is insecure.

Passwords can also be set with the `SET PASSWORD` command. See section [SET OPTION syntax](#).

If you grant privileges for a database, an entry in the `mysql.db` table is created if needed. When all privileges for the database have been removed with `REVOKE`, this entry is deleted.

If a user doesn't have any privileges on a table, the table is not displayed when the user requests a list of tables (e.g., with a `SHOW TABLES` statement).

The `WITH GRANT OPTION` clause gives the user the ability to give to other users any privileges the user has at the specified privilege level. You should be careful to whom you give the **grant** privilege, as two users with different privileges may be able to join privileges!

You cannot grant another user a privilege you don't have yourself; the **grant** privilege allows you to give away only those privileges you possess.

Be aware that when you grant a user the **grant** privilege at a particular privilege level, any privileges the user already possesses (or is given in the future!) at that level are also grantable by that user. Suppose you grant a user the **insert** privilege on a database. If you then grant the **select** privilege on the database and specify `WITH GRANT OPTION`, the user can give away not only the **select** privilege, but also **insert**. If you then grant the **update** privilege to the user on the database, the user can give away the **insert**, **select** and **update**.

You should not grant **alter** privileges to a normal user. If you do that, the user can try to subvert the privilege system by renaming tables!

Note that if you are using table or column privileges for even one user, the server examines table and column privileges for all users and this will slow down **MySQL** a bit.

When `mysqld` starts, all privileges are read into memory. Database, table and column privileges take effect at once and user-level privileges take effect the next time the user connects. Modifications to the grant tables that you perform using `GRANT` or `REVOKE` are noticed by the server immediately. If you modify the grant tables manually (using `INSERT`, `UPDATE`, etc.), you should execute a `FLUSH PRIVILEGES` statement or run `mysqladmin flush-privileges` to tell the server to reload the grant tables. See section [6.9 When privilege changes take effect](#).

The biggest differences between the ANSI SQL and **MySQL** versions of `GRANT` are:

- ANSI SQL doesn't have global or database-level privileges and ANSI SQL doesn't support all privilege types that **MySQL** supports.
- When you drop a table in ANSI SQL, all privileges for the table are revoked. If you revoke a privilege in ANSI SQL, all privileges that were granted based on this privilege are also revoked. In **MySQL**, privileges can be dropped only with explicit `REVOKE` commands or by manipulating the **MySQL** grant tables.

## CREATE INDEX syntax

```
CREATE [UNIQUE] INDEX index_name ON tbl_name (col_name[(length)],... )
```

The `CREATE INDEX` statement doesn't do anything in **MySQL** prior to version 3.22. In 3.22 or later, `CREATE INDEX` is mapped to an `ALTER TABLE` statement to create indexes. See section [ALTER TABLE syntax](#).

Normally, you create all indexes on a table at the time the table itself is created with `CREATE TABLE`. See section [CREATE TABLE syntax](#). `CREATE INDEX` allows you to add indexes to existing tables.

A column list of the form `(col1,col2,...)` creates a multiple-column index. Index values are formed by concatenating the values of the given columns.

For `CHAR` and `VARCHAR` columns, indexes can be created that use only part of a column, using `col_name(length)` syntax. The statement shown below creates an index using the first 10 characters of the `name` column:

```
mysql> CREATE INDEX part_of_name ON customer (name(10));
```

Since most names usually differ in the first 10 characters, this index should not be much slower than an index created from the entire `name` column. Also, using partial columns for indexes can make the index file much smaller, which could save a lot of disk space and might also speed up `INSERT` operations!

For more information about how **MySQL** uses indexes, see section [10.4 How MySQL uses indexes](#).

## DROP INDEX syntax

```
DROP INDEX index_name
```

`DROP INDEX` doesn't do anything in **MySQL** prior to version 3.22. In 3.22 or later, `DROP INDEX` is mapped to an `ALTER TABLE` statement to drop the index. See section [ALTER TABLE syntax](#).

## Comment syntax

The **MySQL** server supports the # to end of line and /\* in-line or multiple-line \*/ comment styles:

```
mysql> select 1+1;      # This comment continues to the end of line
mysql> select 1 /* this is an in-line comment */ + 1;
mysql> select 1+
/*
this is a
multiple-line comment
*/
1;
```

Although the server understands the comment syntax just described, there are some limitations on the way that the `mysql` client parses /\* ... \*/ comments:

- Single-quote and double-quote characters are taken to indicate the beginning of a quoted string, even within a comment. If the quote is not matched by a second quote within the comment, the parser doesn't realize the comment has ended. If you are running `mysql` interactively, you can tell that it has gotten confused like this because the prompt changes from `mysql>` to `'>` or `">`.
- A semicolon is taken to indicate the end of the current SQL statement and anything following it to indicate the beginning of the next statement.

These limitations apply both when you run `mysql` interactively and when you put commands in a file and tell `mysql` to read its input from that file with `mysql < some-file`.

**MySQL** doesn't support the `--` ANSI SQL comment style. See section [5.3.7 '--' as the start of a comment](#).

## CREATE FUNCTION/DROP FUNCTION syntax

```
CREATE FUNCTION function_name RETURNS {STRING|REAL|INTEGER}
        SONAME shared_library_name
```

```
DROP FUNCTION function_name
```

A user-definable function (UDF) is a way to extend **MySQL** with a new function that works like native (built in) **MySQL** functions such as `ABS()` and `CONCAT()`.

`CREATE FUNCTION` saves the function's name, type and shared library name in the `mysql.func` system table. You must have the **insert** and **delete** privileges for the `mysql` database to create and drop functions.

All active functions are reloaded each time the server starts, unless you start `mysqld` with the `--skip-grant-tables` option. In this case, UDF initialization is skipped and UDFs are unavailable. (An active function is one that has been loaded with `CREATE FUNCTION` and not removed with `DROP FUNCTION`.)

For instructions on writing user-definable functions, see section [14 Adding new functions to MySQL](#). For the UDF mechanism to work, functions must be written in C or C++ and your operating system must support dynamic loading.

## Is MySQL picky about reserved words?

A common problem stems from trying to create a table with column names that use the names of datatypes or functions built into **MySQL**, such as `TIMESTAMP` or `GROUP`. You're allowed to do it (for example, `ABS` is an allowed column name), but whitespace is not allowed between a function name and the ``` when using functions whose names are also column names.

The following words are explicitly reserved in **MySQL**. Most of them are forbidden by ANSI SQL92 as column and/or table names (for example, `group`). A few are reserved because **MySQL** needs them and is (currently) using a `yacc` parser:

action	add	all	alter
after	and	as	asc
auto_increment	between	bigint	bit
binary	blob	bool	both
by	cascade	char	character
change	check	column	columns
constraint	create	cross	current_date
current_time	current_timestamp	data	database
databases	date	datetime	day
day_hour	day_minute	day_second	dayofmonth
dayofweek	dayofyear	dec	decimal
default	delete	desc	describe
distinct	distinctrow	double	drop
escaped	enclosed	enum	explain
exists	fields	first	float
float4	float8	foreign	from
for	full	function	grant
group	having	hour	hour_minute
hour_second	ignore	in	index
infile	insert	int	integer
interval	int1	int2	int3
int4	int8	into	if
is	join	key	keys
last_insert_id	leading	left	like
lines	limit	load	lock
long	longblob	longtext	low_priority
match	mediumblob	mediumtext	mediumint
middleint	minute	minute_second	month

monthname	natural	numeric	no
not	null	on	option
optionally	or	order	outer
outfile	partial	password	precision
primary	procedure	processlist	privileges
quarter	read	real	references
rename	regexp	reverse	repeat
replace	restrict	returns	rlike
second	select	set	show
smallint	soname	sql_big_tables	sql_big_selects
sql_select_limit	sql_low_priority_updates	sql_log_off	sql_log_update
straight_join	starting	status	string
table	tables	terminated	text
time	timestamp	tinyblob	tinytext
tinyint	trailing	to	use
using	unique	unlock	unsigned
update	usage	values	varchar
variables	varying	varbinary	with
write	where	year	year_month
zerofill			

The following symbols (from the table above) are disallowed by ANSI SQL but allowed by **MySQL** as column/table names. This is because some of these names are very natural names and a lot of people have already used them.

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