MySQL Press

MySQL AB

# MySQL® Language Reference

*The official guide to the MySQL language and APIs*

# MySQL® Language Reference

# MySQL® Language Reference

MySQL AB

MySQL Press

800 East 96th Street, Indianapolis, Indiana 46240 USA

## MySQL Language Reference

International Standard Book Number: 0-672-32633-7

Library of Congress Catalog Card Number: 2003110975

Printed in the United States of America

First Printing: July 2004

06 05 04 4 3 2 1

## Trademarks

## Warning and Disclaimer

## Bulk Sales

**ASSOCIATE PUBLISHER**
Mark Taber

**ACQUISITIONS EDITOR**
Shelley Johnston

**DEVELOPMENT EDITOR**
Damon Jordan

**MANAGING EDITOR**
Charlotte Clapp

**PROJECT EDITOR**
Andy Beaster

**COPY EDITOR**
Mike Henry

**INDEXER**
Ken Johnson

**PROOFREADER**
Wendy Ott

**PUBLISHING COORDINATOR**
Vanessa Evans

**BOOK DESIGNER**
Gary Adair

**PAGE LAYOUT**
Stacey Richwine-DeRome

**MySQL® Press** is the exclusive publisher of technology books and materials that have been authorized by MySQL AB. MySQL Press books are written and reviewed by the world's leading authorities on MySQL technologies, and are edited, produced, and distributed by the Que/Sams Publishing group of Pearson Education, the worldwide leader in integrated education and computer technology publishing. For more information on MySQL Press and MySQL Press books, please go to **www.mysqlpress.com**.

**MYSQL AB DOCUMENTATION TEAM**
Paul DuBois
Stefan Hinz
Arjen Lentz

**MYSQL HQ**
MySQL AB
Bangårdsgatan 8
S-753 20 Uppsala
Sweden

**UNITED STATES**
MySQL Inc.
2510 Fairview Avenue East
Seattle, WA  98102
USA

**GERMANY, AUSTRIA AND SWITZERLAND**
MySQL GmbH
Schlosserstraße 4
D-72622 Nürtingen
Germany

**FINLAND**
MySQL Finland Oy
Tekniikantie 21
FIN-02150 Espoo
Finland

**FRANCE**
MySQL AB (France)
123, rue du Faubourg St. Antoine
75011, Paris
France

**MySQL® AB** develops, markets, and supports a family of high-performance, affordable database servers and tools. MySQL AB is the sole owner of the MySQL server source code, the MySQL trademark, and the mysql.com domain. For more information on MySQL AB and MySQL AB products, please go to **www.mysql.com** or the following areas of the MySQL Web site:

- Training information: **www.mysql.com/training**
- Support services: **www.mysql.com/support**
- Consulting services: **www.mysql.com/consulting**

# Contents At a Glance

# Table of Contents

## We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

*Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.*

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

| | |
|---|---|
| Email: | mysqlpress@pearsoned.com |
| Mail: | Mark Taber<br>Associate Publisher<br>Pearson Education/MySQL Press<br>800 East 96th Street<br>Indianapolis, IN 46240 USA |

1

# General Information

The MySQL® software delivers a very fast, multi-threaded, multi-user, and robust SQL (Structured Query Language) database server. MySQL Server is intended for mission-critical, heavy-load production systems as well as for embedding into mass-deployed software. MySQL is a registered trademark of MySQL AB.

The MySQL software is Dual Licensed. Users can choose to use the MySQL software as an Open Source/Free Software product under the terms of the GNU General Public License or can purchase a standard commercial license from MySQL AB. See Section 1.4, "MySQL Support and Licensing."

The MySQL Web site (`http://www.mysql.com/`) provides the latest information about the MySQL software.

## 1.1 About This Guide

This guide is made up of those chapters from the *MySQL Reference Manual* that focus on the SQL language used to perform database queries in MySQL. It covers language structure, functions and operators, column types, and SQL statement syntax. A companion guide, the *MySQL Administrator's Guide*, serves as a reference to database administration topics. It covers software installation, server configuration and day-to-day operation, table maintenance, and replication.

This guide is current up to MySQL 5.0.1, but is also applicable for older versions of the MySQL software (such as 3.23 or 4.0-production) because functional changes are indicated with reference to a version number.

Because this guide serves as a reference, it does not provide general instruction on SQL or relational database concepts. It also will not teach you how to use your operating system or command-line interpreter.

The MySQL Database Software is under constant development, and the Reference Manual is updated frequently as well. The most recent version of the manual is available online in searchable form at `http://dev.mysql.com/doc/`. Other formats also are available, including HTML, PDF, and Windows CHM versions.

If you have any suggestions concerning additions or corrections to this manual, please send them to the documentation team at `docs@mysql.com`.

This manual was initially written by David Axmark and Michael "Monty" Widenius. It is now maintained by the MySQL Documentation Team, consisting of Arjen Lentz, Paul DuBois, and Stefan Hinz.

The copyright (2004) to this manual is owned by the Swedish company MySQL AB. See Section 1.4.2, "Copyrights and Licenses Used by MySQL."

## 1.1.1 Conventions Used in This Manual

This manual uses certain typographical conventions:

- `constant`

  Constant-width font is used for command names and options; SQL statements; database, table, and column names; C and Perl code; filenames; URLs; and environment variables. Example: "To see how `mysqladmin` works, invoke it with the `--help` option."

- **`constant bold`**

  Bold constant-width font is used to indicate input that you type in examples.

- *`constant italic`*

  Italic constant-width font is used to indicate variable input for which you should substitute a value of your own choosing.

- '`c`'

  Constant-width font with surrounding quotes is used to indicate character sequences. Example: "To specify a wildcard, use the '`%`' character."

- *italic*

  Italic font is used for emphasis, *like this*.

- **boldface**

  Boldface font is used in table headings and to convey **especially strong emphasis**.

When commands are shown that are meant to be executed from within a particular program, the program is indicated by a prompt shown before the command. For example, `shell>` indicates a command that you execute from your login shell, and `mysql>` indicates a statement that you execute from the `mysql` client program.

```
shell> type a shell command here
mysql> type a mysql statement here
```

The "shell" is your command interpreter. On Unix, this is typically a program such as `sh` or `csh`. On Windows, the equivalent program is `command.com` or `cmd.exe`, typically run in a console window.

When you enter a command or statement shown in an example, do not type the prompt shown in the example.

In example commands, input that you type is indicated in bold type. Variable input for which you should substitute a value that you choose is indicated in italic type. Database, table, and column names must often be substituted into statements. To indicate that such substitution is necessary, this manual uses *db_name*, *tbl_name*, and *col_name*. For example, you might see a statement like this:

```
mysql> SELECT col_name FROM db_name.tbl_name;
```

This means that if you were to enter a similar statement, you would supply your own database, table, and column names, perhaps like this:

```
mysql> SELECT author_name FROM biblio_db.author_list;
```

SQL keywords are not case sensitive and may be written in uppercase or lowercase. This manual uses uppercase.

In syntax descriptions, square brackets ('`[`' and '`]`') are used to indicate optional words or clauses. For example, in the following statement, `IF EXISTS` is optional:

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

When a syntax element consists of a number of alternatives, the alternatives are separated by vertical bars ('`|`'). When one member from a set of choices *may* be chosen, the alternatives are listed within square brackets ('`[`' and '`]`'):

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

When one member from a set of choices *must* be chosen, the alternatives are listed within braces ('`{`' and '`}`'):

```
{DESCRIBE | DESC} tbl_name [col_name | wild]
```

An ellipsis (`...`) indicates the omission of a section of a statement, typically to provide a shorter version of more complex syntax. For example, `INSERT ... SELECT` is shorthand for the form of `INSERT` statement that is followed by a `SELECT` statement.

An ellipsis can also indicate that the preceding syntax element of a statement may be repeated. In the following example, multiple *reset_option* values may be given, with each of those after the first preceded by commas:

```
RESET reset_option [,reset_option] ...
```

Commands for setting shell variables are shown using Bourne shell syntax. For example, the sequence to set an environment variable and run a command looks like this in Bourne shell syntax:

```
shell> VARNAME=value some_command
```

If you are using `csh` or `tcsh`, you must issue commands somewhat differently. You would execute the sequence just shown like this:

```
shell> setenv VARNAME value
shell> some_command
```

## 1.2 Overview of the MySQL Database Management System

MySQL, the most popular Open Source SQL database management system, is developed, distributed, and supported by MySQL AB. MySQL AB is a commercial company, founded by the MySQL developers, that builds its business by providing services around the MySQL database management system. See Section 1.3, "Overview of MySQL AB."

The MySQL Web site (`http://www.mysql.com/`) provides the latest information about MySQL software and MySQL AB.

- MySQL is a database management system.

  A database is a structured collection of data. It may be anything from a simple shopping list to a picture gallery or the vast amounts of information in a corporate network. To add, access, and process data stored in a computer database, you need a database management system such as MySQL Server. Since computers are very good at handling large amounts of data, database management systems play a central role in computing, as standalone utilities or as parts of other applications.
- MySQL is a relational database management system.

  A relational database stores data in separate tables rather than putting all the data in one big storeroom. This adds speed and flexibility. The SQL part of "MySQL" stands for "Structured Query Language." SQL is the most common standardized language used to access databases and is defined by the ANSI/ISO SQL Standard. The SQL standard has been evolving since 1986 and several versions exist. In this manual, "SQL-92" refers to the standard released in 1992, "SQL:1999" refers to the standard released in 1999, and "SQL:2003" refers to the current version of the standard. We use the phrase "the SQL standard" to mean the current version of the SQL Standard at any time.
- MySQL software is Open Source.

  Open Source means that it is possible for anyone to use and modify the software. Anybody can download the MySQL software from the Internet and use it without paying anything. If you wish, you may study the source code and change it to suit your needs. The MySQL software uses the GPL (GNU General Public License) to define what you may and may not do with the software in different situations. If you feel uncomfortable with the GPL or need to embed MySQL code into a commercial application, you can buy a commercially licensed version from us. See Section 1.4.3, "MySQL Licenses."
- The MySQL Database Server is very fast, reliable, and easy to use.

  If that is what you are looking for, you should give it a try. MySQL Server also has a practical set of features developed in close cooperation with our users. You can find a performance comparison of MySQL Server with other database managers at `http://dev.mysql.com/tech-resources/crash-me.php`.

MySQL Server was originally developed to handle large databases much faster than existing solutions and has been successfully used in highly demanding production environments for several years. Although under constant development, MySQL Server today offers a rich and useful set of functions. Its connectivity, speed, and security make MySQL Server highly suited for accessing databases on the Internet.

- MySQL Server works in client/server or embedded systems.

  The MySQL Database Software is a client/server system that consists of a multi-threaded SQL server that supports different backends, several different client programs and libraries, administrative tools, and a wide range of application programming interfaces (APIs).

  We also provide MySQL Server as an embedded multi-threaded library that you can link into your application to get a smaller, faster, easier-to-manage product.

- A large amount of contributed MySQL software is available.

  It is very likely that you will find that your favorite application or language already supports the MySQL Database Server.

The official way to pronounce "MySQL" is "My Ess Que Ell" (not "my sequel"), but we don't mind if you pronounce it as "my sequel" or in some other localized way.

### 1.2.1 History of MySQL

We started out with the intention of using `mSQL` to connect to our tables using our own fast low-level (ISAM) routines. However, after some testing, we came to the conclusion that `mSQL` was not fast enough or flexible enough for our needs. This resulted in a new SQL interface to our database but with almost the same API interface as `mSQL`. This API was designed to allow third-party code that was written for use with `mSQL` to be ported easily for use with MySQL.

The derivation of the name MySQL is not clear. Our base directory and a large number of our libraries and tools have had the prefix "my" for well over 10 years. However, co-founder Monty Widenius's daughter is also named My. Which of the two gave its name to MySQL is still a mystery, even for us.

The name of the MySQL Dolphin (our logo) is "Sakila," which was chosen by the founders of MySQL AB from a huge list of names suggested by users in our "Name the Dolphin" contest. The winning name was submitted by Ambrose Twebaze, an Open Source software developer from Swaziland, Africa. According to Ambrose, the name Sakila has its roots in SiSwati, the local language of Swaziland. Sakila is also the name of a town in Arusha, Tanzania, near Ambrose's country of origin, Uganda.

## 1.2.2 The Main Features of MySQL

The following list describes some of the important characteristics of the MySQL Database Software. See also Section 1.5, "MySQL Development Roadmap," for more information about current and upcoming features.

- Internals and Portability
  - Written in C and C++.
  - Tested with a broad range of different compilers.
  - Works on many different platforms.
  - Uses GNU Automake, Autoconf, and Libtool for portability.
  - APIs for C, C++, Eiffel, Java, Perl, PHP, Python, Ruby, and Tcl are available.
  - Fully multi-threaded using kernel threads. It can easily use multiple CPUs if they are available.
  - Provides transactional and non-transactional storage engines.
  - Uses very fast B-tree disk tables (`MyISAM`) with index compression.
  - Relatively easy to add another storage engine. This is useful if you want to add an SQL interface to an in-house database.
  - A very fast thread-based memory allocation system.
  - Very fast joins using an optimized one-sweep multi-join.
  - In-memory hash tables, which are used as temporary tables.
  - SQL functions are implemented using a highly optimized class library and should be as fast as possible. Usually there is no memory allocation at all after query initialization.
  - The MySQL code is tested both with commercial and Open Source memory leakage detectors.
  - The server is available as a separate program for use in a client/server networked environment. It is also available as a library that can be embedded (linked) into standalone applications. Such applications can be used in isolation or in environments where no network is available.
- Column Types
  - Many column types: signed/unsigned integers 1, 2, 3, 4, and 8 bytes long, `FLOAT`, `DOUBLE`, `CHAR`, `VARCHAR`, `TEXT`, `BLOB`, `DATE`, `TIME`, `DATETIME`, `TIMESTAMP`, `YEAR`, `SET`, `ENUM`, and OpenGIS spatial types.
  - Fixed-length and variable-length records.

- Statements and Functions
  - Full operator and function support in the `SELECT` and `WHERE` clauses of queries. For example:

    ```
    mysql> SELECT CONCAT(first_name, ‘ ‘, last_name)
        -> FROM citizen
        -> WHERE income/dependents > 10000 AND age > 30;
    ```

  - Full support for SQL `GROUP BY` and `ORDER BY` clauses. Support for group functions (`COUNT()`, `COUNT(DISTINCT ...)`, `AVG()`, `STD()`, `SUM()`, `MAX()`, `MIN()`, and `GROUP_CONCAT()`).
  - Support for `LEFT OUTER JOIN` and `RIGHT OUTER JOIN` with both standard SQL and ODBC syntax.
  - Support for aliases on tables and columns as required by standard SQL.
  - `DELETE`, `INSERT`, `REPLACE`, and `UPDATE` return the number of rows that were changed (affected). It is possible to return the number of rows matched instead by setting a flag when connecting to the server.
  - The MySQL-specific `SHOW` command can be used to retrieve information about databases, tables, and indexes. The `EXPLAIN` command can be used to determine how the optimizer resolves a query.
  - Function names do not clash with table or column names. For example, `ABS` is a valid column name. The only restriction is that for a function call, no spaces are allowed between the function name and the ‘`(`’ that follows it.
  - You can mix tables from different databases in the same query (as of MySQL 3.22).
- Security
  - A privilege and password system that is very flexible and secure, and that allows host-based verification. Passwords are secure because all password traffic is encrypted when you connect to a server.
- Scalability and Limits
  - Handles large databases. We use MySQL Server with databases that contain 50 million records. We also know of users who use MySQL Server with 60,000 tables and about 5,000,000,000 rows.
  - Up to 64 indexes per table are allowed (32 before MySQL 4.1.2). Each index may consist of 1 to 16 columns or parts of columns. The maximum index width is 1000 bytes (500 before MySQL 4.1.2). An index may use a prefix of a column for `CHAR`, `VARCHAR`, `BLOB`, or `TEXT` column types.
- Connectivity
  - Clients can connect to the MySQL server using TCP/IP sockets on any platform. On Windows systems in the NT family (NT, 2000, or XP), clients can connect using named pipes. On Unix systems, clients can connect using Unix domain socket files.

  - The Connector/ODBC interface provides MySQL support for client programs that use ODBC (Open Database Connectivity) connections. For example, you can use MS Access to connect to your MySQL server. Clients can be run on Windows or Unix. Connector/ODBC source is available. All ODBC 2.5 functions are supported, as are many others.
  - The Connector/JDBC interface provides MySQL support for Java client programs that use JDBC connections. Clients can be run on Windows or Unix. Connector/JDBC source is available.
- Localization
  - The server can provide error messages to clients in many languages.
  - Full support for several different character sets, including `latin1` (ISO-8859-1), `german`, `big5`, `ujis`, and more. For example, the Scandinavian characters 'å', 'ä' and 'ö' are allowed in table and column names. Unicode support is available as of MySQL 4.1.
  - All data is saved in the chosen character set. All comparisons for normal string columns are case-insensitive.
  - Sorting is done according to the chosen character set (using Swedish collation by default). It is possible to change this when the MySQL server is started. To see an example of very advanced sorting, look at the Czech sorting code. MySQL Server supports many different character sets that can be specified at compile time and runtime.
- Clients and Tools
  - The MySQL server has built-in support for SQL statements to check, optimize, and repair tables. These statements are available from the command line through the `mysqlcheck` client. MySQL also includes `myisamchk`, a very fast command-line utility for performing these operations on `MyISAM` tables.
  - All MySQL programs can be invoked with the `--help` or `-?` options to obtain online assistance.

## 1.2.3 MySQL Stability

This section addresses the questions, "*How stable is MySQL Server?*" and, "*Can I depend on MySQL Server in this project?*" We will try to clarify these issues and answer some important questions that concern many potential users. The information in this section is based on data gathered from the mailing lists, which are very active in identifying problems as well as reporting types of use.

The original code stems back to the early 1980s. It provides a stable code base, and the `ISAM` table format used by the original storage engine remains backward-compatible. At

TcX, the predecessor of MySQL AB, MySQL code has worked in projects since mid-1996, without any problems. When the MySQL Database Software initially was released to a wider public, our new users quickly found some pieces of untested code. Each new release since then has had fewer portability problems, even though each new release has also had many new features.

Each release of the MySQL Server has been usable. Problems have occurred only when users try code from the "gray zones." Naturally, new users don't know what the gray zones are; this section therefore attempts to document those areas that are currently known. The descriptions mostly deal with Version 3.23 and 4.0 of MySQL Server. All known and reported bugs are fixed in the latest version, with the exception of those listed in the bugs section, which are design-related. See Section 1.8.7, "Known Errors and Design Deficiencies in MySQL."

The MySQL Server design is multi-layered with independent modules. Some of the newer modules are listed here with an indication of how well-tested each of them is:

- Replication (Gamma)

  Large groups of servers using replication are in production use, with good results. Work on enhanced replication features is continuing in MySQL 5.x.
- `InnoDB` tables (Stable)

  The `InnoDB` transactional storage engine has been declared stable in the MySQL 3.23 tree, starting from Version 3.23.49. `InnoDB` is being used in large, heavy-load production systems.
- `BDB` tables (Gamma)

  The `Berkeley DB` code is very stable, but we are still improving the `BDB` transactional storage engine interface in MySQL Server, so it will take some time before this is as well tested as the other table types.
- Full-text searches (Beta)

  Full-text searching works but is not yet widely used. Important enhancements have been implemented in MySQL 4.0.
- `Connector/ODBC` 3.51 (Stable)

  `Connector/ODBC` 3.51 uses ODBC SDK 3.51 and is in wide production use. Some issues brought up appear to be application-related and independent of the ODBC driver or underlying database server.
- Automatic recovery of `MyISAM` tables (Gamma)

  This status applies only to the new code in the `MyISAM` storage engine that checks when opening a table whether it was closed properly and executes an automatic check or repair of the table if it wasn't.

## 1.2.4 How Big MySQL Tables Can Be

MySQL 3.22 had a 4GB (4 gigabyte) limit on table size. With the `MyISAM` storage engine in MySQL 3.23, the maximum table size was increased to 8 million terabytes ($2^{63}$ bytes). With this larger allowed table size, the maximum effective table size for MySQL databases now usually is determined by operating system constraints on file sizes, not by MySQL internal limits.

The `InnoDB` storage engine maintains `InnoDB` tables within a tablespace that can be created from several files. This allows a table to exceed the maximum individual file size. The tablespace can include raw disk partitions, which allows extremely large tables. The maximum tablespace size is 64TB.

The following table lists some examples of operating system file-size limits:

| **Operating System** | **File Size Limit** |
|---|---|
| Linux-Intel 32-bit | 2GB, much more when using LFS |
| Linux-Alpha | 8TB (?) |
| Solaris 2.5.1 | 2GB (4GB possible with patch) |
| Solaris 2.6 | 4GB (can be changed with flag) |
| Solaris 2.7 Intel | 4GB |
| Solaris 2.7 UltraSPARC | 512GB |
| NetWare w/NSS filesystem | 8TB |

On Linux 2.2, you can get `MyISAM` tables larger than 2GB in size by using the Large File Support (LFS) patch for the ext2 filesystem. On Linux 2.4, patches also exist for ReiserFS to get support for big files. Most current Linux distributions are based on kernel 2.4 and already include all the required LFS patches. However, the maximum available file size still depends on several factors, one of them being the filesystem used to store MySQL tables.

For a detailed overview about LFS in Linux, have a look at Andreas Jaeger's "Large File Support in Linux" page at `http://www.suse.de/~aj/linux_lfs.html`.

By default, MySQL creates `MyISAM` tables with an internal structure that allows a maximum size of about 4GB. You can check the maximum table size for a table with the `SHOW TABLE STATUS` statement or with `myisamchk -dv` *`tbl_name`*.

If you need a `MyISAM` table that will be larger than 4GB in size (and your operating system supports large files), the `CREATE TABLE` statement allows `AVG_ROW_LENGTH` and `MAX_ROWS` options. You can also change these options with `ALTER TABLE` after the table has been created, to increase the table's maximum allowable size.

Other ways to work around file-size limits for `MyISAM` tables are as follows:

- If your large table is read-only, you can use `myisampack` to compress it. `myisampack` usually compresses a table by at least 50%, so you can have, in effect, much bigger tables. `myisampack` also can merge multiple tables into a single table.
- Another way to get around the operating system file limit for `MyISAM` data files is by using the `RAID` options.

- MySQL includes a `MERGE` library that allows you to handle a collection of `MyISAM` tables that have identical structure as a single `MERGE` table.

### 1.2.5 Year 2000 Compliance

The MySQL Server itself has no problems with Year 2000 (Y2K) compliance:

- MySQL Server uses Unix time functions that handle dates into the year `2037` for `TIMESTAMP` values. For `DATE` and `DATETIME` values, dates through the year `9999` are accepted.
- All MySQL date functions are implemented in one source file, `sql/time.cc`, and are coded very carefully to be year 2000-safe.
- In MySQL 3.22 and later, the `YEAR` column type can store years `0` and `1901` to `2155` in one byte and display them using two or four digits. All two-digit years are considered to be in the range `1970` to `2069`, which means that if you store `01` in a `YEAR` column, MySQL Server treats it as `2001`.

The following simple demonstration illustrates that MySQL Server has no problems with `DATE` or `DATETIME` values through the year 9999, and no problems with `TIMESTAMP` values until after the year 2030:

```
mysql> DROP TABLE IF EXISTS y2k;
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE y2k (date DATE,
    ->                   date_time DATETIME,
    ->                   time_stamp TIMESTAMP);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO y2k VALUES
    -> ('1998-12-31','1998-12-31 23:59:59',19981231235959),
    -> ('1999-01-01','1999-01-01 00:00:00',19990101000000),
    -> ('1999-09-09','1999-09-09 23:59:59',19990909235959),
    -> ('2000-01-01','2000-01-01 00:00:00',20000101000000),
    -> ('2000-02-28','2000-02-28 00:00:00',20000228000000),
    -> ('2000-02-29','2000-02-29 00:00:00',20000229000000),
    -> ('2000-03-01','2000-03-01 00:00:00',20000301000000),
    -> ('2000-12-31','2000-12-31 23:59:59',20001231235959),
    -> ('2001-01-01','2001-01-01 00:00:00',20010101000000),
    -> ('2004-12-31','2004-12-31 23:59:59',20041231235959),
    -> ('2005-01-01','2005-01-01 00:00:00',20050101000000),
    -> ('2030-01-01','2030-01-01 00:00:00',20300101000000),
    -> ('2040-01-01','2040-01-01 00:00:00',20400101000000),
    -> ('9999-12-31','9999-12-31 23:59:59',99991231235959);
Query OK, 14 rows affected (0.01 sec)
Records: 14  Duplicates: 0  Warnings: 2
```

```
mysql> SELECT * FROM y2k;
+------------+---------------------+----------------+
| date       | date_time           | time_stamp     |
+------------+---------------------+----------------+
| 1998-12-31 | 1998-12-31 23:59:59 | 19981231235959 |
| 1999-01-01 | 1999-01-01 00:00:00 | 19990101000000 |
| 1999-09-09 | 1999-09-09 23:59:59 | 19990909235959 |
| 2000-01-01 | 2000-01-01 00:00:00 | 20000101000000 |
| 2000-02-28 | 2000-02-28 00:00:00 | 20000228000000 |
| 2000-02-29 | 2000-02-29 00:00:00 | 20000229000000 |
| 2000-03-01 | 2000-03-01 00:00:00 | 20000301000000 |
| 2000-12-31 | 2000-12-31 23:59:59 | 20001231235959 |
| 2001-01-01 | 2001-01-01 00:00:00 | 20010101000000 |
| 2004-12-31 | 2004-12-31 23:59:59 | 20041231235959 |
| 2005-01-01 | 2005-01-01 00:00:00 | 20050101000000 |
| 2030-01-01 | 2030-01-01 00:00:00 | 20300101000000 |
| 2040-01-01 | 2040-01-01 00:00:00 | 00000000000000 |
| 9999-12-31 | 9999-12-31 23:59:59 | 00000000000000 |
+------------+---------------------+----------------+
14 rows in set (0.00 sec)
```

The final two `TIMESTAMP` column values are zero because the final year values (`2040`, `9999`) exceed the `TIMESTAMP` maximum. The `TIMESTAMP` data type, which is used to store the current time, supports values that range from `19700101000000` to `20300101000000` on 32-bit machines (signed value). On 64-bit machines, `TIMESTAMP` handles values up to `2106` (unsigned value).

Although MySQL Server itself is Y2K-safe, you may run into problems if you use it with applications that are not Y2K-safe. For example, many old applications store or manipulate years using two-digit values (which are ambiguous) rather than four-digit values. This problem may be compounded by applications that use values such as `00` or `99` as "missing" value indicators. Unfortunately, these problems may be difficult to fix because different applications may be written by different programmers, each of whom may use a different set of conventions and date-handling functions.

Thus, even though MySQL Server has no Y2K problems, it is the application's responsibility to provide unambiguous input.

## 1.3 Overview of MySQL AB

MySQL AB is the company of the MySQL founders and main developers. MySQL AB was originally established in Sweden by David Axmark, Allan Larsson, and Michael "Monty" Widenius.

The developers of the MySQL server are all employed by the company. We are a virtual organization with people in a dozen countries around the world. We communicate extensively over the Internet every day with one another and with our users, supporters, and partners.

We are dedicated to developing the MySQL database software and promoting it to new users. MySQL AB owns the copyright to the MySQL source code, the MySQL logo and trademark, and this manual. See Section 1.2, "Overview of the MySQL Database Management System."

The MySQL core values show our dedication to MySQL and Open Source.

These core values direct how MySQL AB works with the MySQL server software:

- To be the best and the most widely used database in the world
- To be available and affordable by all
- To be easy to use
- To be continuously improved while remaining fast and safe
- To be fun to use and improve
- To be free from bugs

These are the core values of the company MySQL AB and its employees:

- We subscribe to the Open Source philosophy and support the Open Source community
- We aim to be good citizens
- We prefer partners that share our values and mindset
- We answer email and provide support
- We are a virtual company, networking with others
- We work against software patents

The MySQL Web site (`http://www.mysql.com/`) provides the latest information about MySQL and MySQL AB.

By the way, the "AB" part of the company name is the acronym for the Swedish "aktiebolag," or "stock company." It translates to "MySQL, Inc." In fact, MySQL, Inc. and MySQL GmbH are examples of MySQL AB subsidiaries. They are located in the United States and Germany, respectively.

### 1.3.1 The Business Model and Services of MySQL AB

One of the most common questions we encounter is, "*How can you make a living from something you give away for free?*" This is how:

- MySQL AB makes money on support, services, commercial licenses, and royalties.
- We use these revenues to fund product development and to expand the MySQL business.

The company has been profitable since its inception. In October 2001, we accepted venture financing from leading Scandinavian investors and a handful of business angels. This investment is used to solidify our business model and build a basis for sustainable growth.

### 1.3.1.1 Support

MySQL AB is run and owned by the founders and main developers of the MySQL database. The developers are committed to providing support to customers and other users in order to stay in touch with their needs and problems. All our support is provided by qualified developers. Really tricky questions are answered by Michael "Monty" Widenius, principal author of the MySQL Server.

Paying customers receive high-quality support directly from MySQL AB. MySQL AB also provides the MySQL mailing lists as a community resource where anyone may ask questions.

For more information and ordering support at various levels, see Section 1.4, "MySQL Support and Licensing."

### 1.3.1.2 Training and Certification

MySQL AB delivers MySQL and related training worldwide. We offer both open courses and in-house courses tailored to the specific needs of your company. MySQL Training is also available through our partners, the Authorized MySQL Training Centers.

Our training material uses the same sample databases used in our documentation and our sample applications, and is always updated to reflect the latest MySQL version. Our trainers are backed by the development team to guarantee the quality of the training and the continuous development of the course material. This also ensures that no questions raised during the courses remain unanswered.

Attending our training courses will enable you to achieve your MySQL application goals. You will also:

- Save time
- Improve the performance of your applications
- Reduce or eliminate the need for additional hardware, decreasing cost
- Enhance security
- Increase customer and co-worker satisfaction
- Prepare yourself for MySQL Certification

If you are interested in our training as a potential participant or as a training partner, please visit the training section at `http://www.mysql.com/training/`, or send email to `training@mysql.com`.

For details about the MySQL Certification Program, please see `http://www.mysql.com/certification/`.

### 1.3.1.3 Consulting

MySQL AB and its Authorized Partners offer consulting services to users of MySQL Server and to those who embed MySQL Server in their own software, all over the world.

Our consultants can help you design and tune your databases, construct efficient queries, tune your platform for optimal performance, resolve migration issues, set up replication, build robust transactional applications, and more. We also help customers embed MySQL Server in their products and applications for large-scale deployment.

Our consultants work in close collaboration with our development team, which ensures the technical quality of our professional services. Consulting assignments range from two-day power-start sessions to projects that span weeks and months. Our expertise covers not only MySQL Server, it also extends into programming and scripting languages such as PHP, Perl, and more.

If you are interested in our consulting services or want to become a consulting partner, please visit the consulting section of our Web site at `http://www.mysql.com/consulting/` or contact our consulting staff at `consulting@mysql.com`.

### 1.3.1.4 Commercial Licenses

The MySQL database is released under the GNU General Public License (GPL). This means that the MySQL software can be used free of charge under the GPL. If you do not want to be bound by the GPL terms (such as the requirement that your application must also be GPL), you may purchase a commercial license for the same product from MySQL AB; see `https://order.mysql.com/`. Since MySQL AB owns the copyright to the MySQL source code, we are able to employ Dual Licensing, which means that the same product is available under GPL and under a commercial license. This does not in any way affect the Open Source commitment of MySQL AB. For details about when a commercial license is required, please see Section 1.4.3, "MySQL Licenses."

We also sell commercial licenses of third-party Open Source GPL software that adds value to MySQL Server. A good example is the `InnoDB` transactional storage engine that offers `ACID` support, row-level locking, crash recovery, multi-versioning, foreign key support, and more.

### 1.3.1.5 Partnering

MySQL AB has a worldwide partner program that covers training courses, consulting and support, publications, plus reselling and distributing MySQL and related products. MySQL AB Partners get visibility on the `http://www.mysql.com/` Web site and the right to use special versions of the MySQL trademarks to identify their products and promote their business.

If you are interested in becoming a MySQL AB Partner, please email `partner@mysql.com`.

The word MySQL and the MySQL dolphin logo are trademarks of MySQL AB. See Section 1.4.4, “MySQL AB Logos and Trademarks.” These trademarks represent a significant value that the MySQL founders have built over the years.

The MySQL Web site (`http://www.mysql.com/`) is popular among developers and users. In December 2003, we served 16 million page views. Our visitors represent a group that makes purchase decisions and recommendations for both software and hardware. 12% of our visitors authorize purchase decisions, and only 9% have no involvement at all in purchase decisions. More than 65% have made one or more online business purchases within the last half-year, and 70% plan to make one in the next few months.

## 1.3.2 Contact Information

The MySQL Web site (`http://www.mysql.com/`) provides the latest information about MySQL and MySQL AB.

For press services and inquiries not covered in our news releases (`http://www.mysql.com/news-and-events/`), please send email to `press@mysql.com`.

If you have a support contract with MySQL AB, you will get timely, precise answers to your technical questions about the MySQL software. For more information, see Section 1.4.1, “Support Offered by MySQL AB.” On our Web site, see `http://www.mysql.com/support/`, or send email to `sales@mysql.com`.

For information about MySQL training, please visit the training section at `http://www.mysql.com/training/`, or send email to `training@mysql.com`. See Section 1.3.1.2, “Training and Certification.”

For information on the MySQL Certification Program, please see `http://www.mysql.com/certification/`. See Section 1.3.1.2, “Training and Certification.”

If you’re interested in consulting, please visit the consulting section of our Web site at `http://www.mysql.com/consulting/`, or send email to `consulting@mysql.com`. See Section 1.3.1.3, “Consulting.”

Commercial licenses may be purchased online at `https://order.mysql.com/`. There you will also find information on how to fax your purchase order to MySQL AB. More information about licensing can be found at `http://www.mysql.com/products/pricing.html`. If you have questions regarding licensing or you want a quote for high-volume licensing, please fill in the contact form on our Web site (`http://www.mysql.com/`), or send email to `licensing@mysql.com` (for licensing questions) or to `sales@mysql.com` (for sales inquiries). See Section 1.4.3, “MySQL Licenses.”

If you represent a business that is interested in partnering with MySQL AB, please send email to `partner@mysql.com`. See Section 1.3.1.5, “Partnering.”

For more information on the MySQL trademark policy, refer to `http://www.mysql.com/company/trademark.html`, or send email to `trademark@mysql.com`. See Section 1.4.4, “MySQL AB Logos and Trademarks.”

If you are interested in any of the MySQL AB jobs listed in our jobs section (`http://www.mysql.com/company/jobs/`), please send email to `jobs@mysql.com`. Please do not send your CV as an attachment, but rather as plain text at the end of your email message.

For general discussion among our many users, please direct your attention to the appropriate mailing list. See Section 1.7.1, "MySQL Mailing Lists."

Reports of errors (often called "bugs"), as well as questions and comments, should be sent to the general MySQL mailing list. See Section 1.7.1.1, "The MySQL Mailing Lists." If you have found a sensitive security bug in MySQL Server, please let us know immediately by sending email to `security@mysql.com`. See Section 1.7.1.3, "How to Report Bugs or Problems."

If you have benchmark results that we can publish, please contact us via email at `benchmarks@mysql.com`.

If you have suggestions concerning additions or corrections to this manual, please send them to the documentation team via email at `docs@mysql.com`.

For questions or comments about the workings or content of the MySQL Web site (`http://www.mysql.com/`), please send email to `webmaster@mysql.com`.

MySQL AB has a privacy policy, which can be read at `http://www.mysql.com/company/privacy.html`. For any queries regarding this policy, please send email to `privacy@mysql.com`.

For all other inquiries, please send email to `info@mysql.com`.

## 1.4 MySQL Support and Licensing

This section describes MySQL support and licensing arrangements.

### 1.4.1 Support Offered by MySQL AB

Technical support from MySQL AB means individualized answers to your unique problems direct from the software engineers who code the MySQL database engine.

We try to take a broad and inclusive view of technical support. Almost any problem involving MySQL software is important to us if it's important to you. Typically, customers seek help on how to get different commands and utilities to work, remove performance bottlenecks, restore crashed systems, understand the impact of operating system or networking issues on MySQL, set up best practices for backup and recovery, utilize APIs, and so on. Our support covers only the MySQL server and our own utilities, not third-party products that access the MySQL server, although we try to help with these where we can.

Detailed information about our various support options is given at `http://www.mysql.com/support/`, where support contracts can also be ordered online. To contact our sales staff, send email to `sales@mysql.com`.

Technical support is like life insurance. You can live happily without it for years. However, when your hour arrives, it becomes critically important, but it's too late to buy it. If you use MySQL Server for important applications and encounter sudden difficulties, it may be too time-consuming to figure out all the answers yourself. You may need immediate access to the most experienced MySQL troubleshooters available, those employed by MySQL AB.

## 1.4.2 Copyrights and Licenses Used by MySQL

MySQL AB owns the copyright to the MySQL source code, the MySQL logos and trademarks, and this manual. See Section 1.3, "Overview of MySQL AB." Several different licenses are relevant to the MySQL distribution:

1. All the MySQL-specific source in the server, the `mysqlclient` library and the client, as well as the GNU `readline` library, are covered by the GNU General Public License. See `http://www.fsf.org/licenses/`. The text of this license can be found as the file `COPYING` in MySQL distributions.
2. The GNU `getopt` library is covered by the GNU Lesser General Public License. See `http://www.fsf.org/licenses/`.
3. Some parts of the source (the `regexp` library) are covered by a Berkeley-style copyright.
4. Older versions of MySQL (3.22 and earlier) are subject to a stricter license (`http://www.mysql.com/products/mypl.html`). See the documentation of the specific version for information.
5. The *MySQL Reference Manual* is *not* distributed under a GPL-style license. Use of the manual is subject to the following terms:
   - Conversion to other formats is allowed, but the actual content may not be altered or edited in any way.
   - You may create a printed copy for your own personal use.
   - For all other uses, such as selling printed copies or using (parts of) the manual in another publication, prior written agreement from MySQL AB is required.

   Please send an email message to `docs@mysql.com` for more information or if you are interested in doing a translation.

For information about how the MySQL licenses work in practice, please refer to Section 1.4.3, "MySQL Licenses," and Section 1.4.4, "MySQL AB Logos and Trademarks."

## 1.4.3 MySQL Licenses

The MySQL software is released under the GNU General Public License (GPL), which is probably the best known Open Source license. The formal terms of the GPL license can be found at `http://www.fsf.org/licenses/`. See also `http://www.fsf.org/licenses/gpl-faq.html` and `http://www.gnu.org/philosophy/enforcing-gpl.html`.

Our GPL licensing is supported by an optional license exception that enables many Free/Libre and Open Source Software ("FLOSS") applications to include the GPL-licensed MySQL client libraries despite the fact that not all FLOSS licenses are compatible with the GPL. For details, see `http://www.mysql.com/products/licensing/foss-exception.html`.

Because the MySQL software is released under the GPL, it may often be used for free, but for certain uses you may want or need to buy commercial licenses from MySQL AB at `https://order.mysql.com/`. See `http://www.mysql.com/products/licensing.html` for more information.

Older versions of MySQL (3.22 and earlier) are subject to a stricter license (`http://www.mysql.com/products/mypl.html`). See the documentation of the specific version for information.

Please note that the use of the MySQL software under commercial license, GPL, or the old MySQL license does not automatically give you the right to use MySQL AB trademarks. See Section 1.4.4, "MySQL AB Logos and Trademarks."

#### 1.4.3.1 Using the MySQL Software Under a Commercial License

The GPL license is contagious in the sense that when a program is linked to a GPL program, all the source code for all the parts of the resulting product must also be released under the GPL. If you do not follow this GPL requirement, you break the license terms and forfeit your right to use the GPL program altogether. You also risk damages.

You need a commercial license under these conditions:

- When you link a program with any GPL code from the MySQL software and don't want the resulting product to be licensed under GPL, perhaps because you want to build a commercial product or keep the added non-GPL code closed source for other reasons. When purchasing commercial licenses, you are not using the MySQL software under GPL even though it's the same code.
- When you distribute a non-GPL application that works *only* with the MySQL software and ship it with the MySQL software. This type of solution is considered to be linking even if it's done over a network.
- When you distribute copies of the MySQL software without providing the source code as required under the GPL license.
- When you want to support the further development of the MySQL database even if you don't formally need a commercial license. Purchasing support directly from MySQL AB is another good way of contributing to the development of the MySQL software, with immediate advantages for you. See Section 1.4.1, "Support Offered by MySQL AB."

Our GPL licensing is supported by an optional license exception that enables many Free/Libre and Open Source Software ("FLOSS") applications to include the GPL-licensed MySQL client libraries despite the fact that not all FLOSS licenses are compatible with the GPL. For details, see `http://www.mysql.com/products/licensing/foss-exception.html`.

If you require a commercial license, you will need one for each installation of the MySQL software. This covers any number of CPUs on a machine, and there is no artificial limit on the number of clients that connect to the server in any way.

For commercial licenses, please visit our Web site at `http://www.mysql.com/products/licensing.html`. For support contracts, see `http://www.mysql.com/support/`. If you have special needs, please contact our sales staff via email at `sales@mysql.com`.

### 1.4.3.2 Using the MySQL Software for Free Under GPL

You can use the MySQL software for free under the GPL if you adhere to the conditions of the GPL. For additional details about the GPL, including answers to common questions, see the generic FAQ from the Free Software Foundation at `http://www.fsf.org/licenses/gpl-faq.html`.

Our GPL licensing is supported by an optional license exception that enables many Free/Libre and Open Source Software ("FLOSS") applications to include the GPL-licensed MySQL client libraries despite the fact that not all FLOSS licenses are compatible with the GPL. For details, see `http://www.mysql.com/products/licensing/foss-exception.html`.

Common uses of the GPL include:

- When you distribute both your own application and the MySQL source code under the GPL with your product.
- When you distribute the MySQL source code bundled with other programs that are not linked to or dependent on the MySQL system for their functionality even if you sell the distribution commercially. This is called "mere aggregation" in the GPL license.
- When you are not distributing *any* part of the MySQL system, you can use it for free.
- When you are an Internet Service Provider (ISP), offering Web hosting with MySQL servers for your customers. We encourage people to use ISPs that have MySQL support, because doing so will give them the confidence that their ISP will, in fact, have the resources to solve any problems they may experience with the MySQL installation. Even if an ISP does not have a commercial license for MySQL Server, their customers should at least be given read access to the source of the MySQL installation so that the customers can verify that it is correctly patched.
- When you use the MySQL database software in conjunction with a Web server, you do not need a commercial license (so long as it is not a product you distribute). This is true even if you run a commercial Web server that uses MySQL Server, because you are not distributing any part of the MySQL system. However, in this case we would like you to purchase MySQL support because the MySQL software is helping your enterprise.

If your use of MySQL database software does not require a commercial license, we encourage you to purchase support from MySQL AB anyway. This way you contribute toward MySQL development and also gain immediate advantages for yourself. See Section 1.4.1, "Support Offered by MySQL AB."

If you use the MySQL database software in a commercial context such that you profit by its use, we ask that you further the development of the MySQL software by purchasing some level of support. We feel that if the MySQL database helps your business, it is reasonable to ask that you help MySQL AB. (Otherwise, if you ask us support questions, you are not only using for free something into which we've put a lot a work, you're asking us to provide free support, too.)

## 1.4.4 MySQL AB Logos and Trademarks

Many users of the MySQL database want to display the MySQL AB dolphin logo on their Web sites, books, or boxed products. We welcome and encourage this, although it should be noted that the word MySQL and the MySQL dolphin logo are trademarks of MySQL AB and may only be used as stated in our trademark policy at `http://www.mysql.com/company/trademark.html`.

### 1.4.4.1 The Original MySQL Logo

The MySQL dolphin logo was designed by the Finnish advertising agency Priority in 2001. The dolphin was chosen as a suitable symbol for the MySQL database management system, which is like a smart, fast, and lean animal, effortlessly navigating oceans of data. We also happen to like dolphins.

The original MySQL logo may only be used by representatives of MySQL AB and by those having a written agreement allowing them to do so.

### 1.4.4.2 MySQL Logos That May Be Used Without Written Permission

We have designed a set of special *Conditional Use* logos that may be downloaded from our Web site at `http://www.mysql.com/press/logos.html` and used on third-party Web sites without written permission from MySQL AB. The use of these logos is not entirely unrestricted but, as the name implies, subject to our trademark policy that is also available on our Web site. You should read through the trademark policy if you plan to use them. The requirements are basically as follows:

- Use the logo you need as displayed on the `http://www.mysql.com/` site. You may scale it to fit your needs, but may not change colors or design, or alter the graphics in any way.
- Make it evident that you, and not MySQL AB, are the creator and owner of the site that displays the MySQL trademark.
- Don't use the trademark in a way that is detrimental to MySQL AB or to the value of MySQL AB trademarks. We reserve the right to revoke the right to use the MySQL AB trademark.
- If you use the trademark on a Web site, make it clickable, leading directly to `http://www.mysql.com/`.
- If you use the MySQL database under GPL in an application, your application must be Open Source and must be able to connect to a MySQL server.

Contact us via email at `trademark@mysql.com` to inquire about special arrangements to fit your needs.

### 1.4.4.3 When You Need Written Permission to Use MySQL Logos

You need written permission from MySQL AB before using MySQL logos in the following cases:

- When displaying any MySQL AB logo anywhere except on your Web site.
- When displaying any MySQL AB logo except the *Conditional Use* logos (mentioned previously) on Web sites or elsewhere.

Due to legal and commercial reasons, we monitor the use of MySQL trademarks on products, books, and other items. We usually require a fee for displaying MySQL AB logos on commercial products, since we think it is reasonable that some of the revenue is returned to fund further development of the MySQL database.

### 1.4.4.4 MySQL AB Partnership Logos

MySQL partnership logos may be used only by companies and persons having a written partnership agreement with MySQL AB. Partnerships include certification as a MySQL trainer or consultant. For more information, please see Section 1.3.1.5, "Partnering."

### 1.4.4.5 Using the Word MySQL in Printed Text or Presentations

MySQL AB welcomes references to the MySQL database, but it should be noted that the word MySQL is a trademark of MySQL AB. Because of this, you must append the trademark notice symbol (®) to the first or most prominent use of the word MySQL in a text and, where appropriate, state that MySQL is a trademark of MySQL AB. For more information, please refer to our trademark policy at `http://www.mysql.com/company/trademark.html`.

### 1.4.4.6 Using the Word MySQL in Company and Product Names

Use of the word MySQL in company or product names or in Internet domain names is not allowed without written permission from MySQL AB.

## 1.5 MySQL Development Roadmap

This section provides a snapshot of the MySQL development roadmap, including major features implemented or planned for MySQL 4.0, 4.1, 5.0, and 5.1. The following sections provide information for each release series.

The production release series is MySQL 4.0, which was declared stable for production use as of Version 4.0.12, released in March 2003. This means that future 4.0 development will be limited only to making bug fixes. For the older MySQL 3.23 series, only critical bug fixes will be made.

Active MySQL development currently is taking place in the MySQL 4.1 and 5.0 release series. This means that new features are being added to MySQL 4.1 and MySQL 5.0. 4.1 is available in beta status, and 5.0 is avalable in alpha status.

Plans for some of the most requested features are summarized in the following table:

| **Feature** | **MySQL Series** |
|---|---|
| Unions | 4.0 |
| Subqueries | 4.1 |
| R-trees | 4.1 (for `MyISAM` tables) |
| Stored procedures | 5.0 |
| Views | 5.0 |
| Cursors | 5.0 |
| Foreign keys | 5.1 (already implemented in 3.23 for `InnoDB`) |
| Triggers | 5.1 |
| Full outer join | 5.1 |
| Constraints | 5.1 |

## 1.5.1 MySQL 4.0 in a Nutshell

Long awaited by our users, MySQL Server 4.0 is now available in production status.

MySQL 4.0 is available for download at `http://dev.mysql.com/` and from our mirrors. MySQL 4.0 has been tested by a large number of users and is in production use at many large sites.

The major new features of MySQL Server 4.0 are geared toward our existing business and community users, enhancing the MySQL database software as the solution for mission-critical, heavy-load database systems. Other new features target the users of embedded databases.

### 1.5.1.1 Features Available in MySQL 4.0

- Speed enhancements
  - MySQL 4.0 has a query cache that can give a huge speed boost to applications with repetitive queries.
  - Version 4.0 further increases the speed of MySQL Server in a number of areas, such as bulk `INSERT` statements, searching on packed indexes, full-text searching (using `FULLTEXT` indexes), and `COUNT(DISTINCT)`.
- Embedded MySQL Server introduced
  - The new Embedded Server library can easily be used to create standalone and embedded applications. The embedded server provides an alternative to using MySQL in a client/server environment.
- `InnoDB` storage engine as standard
  - The `InnoDB` storage engine is now offered as a standard feature of the MySQL server. This means full support for ACID transactions, foreign keys with cascading `UPDATE` and `DELETE`, and row-level locking are now standard features.

- New functionality
  - The enhanced `FULLTEXT` search properties of MySQL Server 4.0 enables `FULLTEXT` indexing of large text masses with both binary and natural-language searching logic. You can customize minimal word length and define your own stop word lists in any human language, enabling a new set of applications to be built with MySQL Server.
- Standards compliance, portability, and migration
  - Many users will also be happy to learn that MySQL Server now supports the `UNION` statement, a long-awaited standard SQL feature.
  - MySQL now runs natively on the Novell NetWare platform beginning with NetWare 6.0.
  - Features to simplify migration from other database systems to MySQL Server include `TRUNCATE TABLE` (as in Oracle).
- Internationalization
  - Our German, Austrian, and Swiss users will note that MySQL 4.0 now supports a new character set, `latin1_de`, which ensures that the *German sorting order* sorts words with umlauts in the same order as do German telephone books.
- Usability enhancements

  In the process of implementing features for new users, we have not forgotten requests from our loyal community of existing users.
  - Most `mysqld` parameters (startup options) can now be set without taking down the server. This is a convenient feature for database administrators (DBAs).
  - Multiple-table `DELETE` and `UPDATE` statements have been added.
  - On Windows, symbolic link handling at the database level is enabled by default. On Unix, the `MyISAM` storage engine now supports symbolic linking at the table level (and not just the database level as before).
  - `SQL_CALC_FOUND_ROWS` and `FOUND_ROWS()` are new functions that make it possible to find out the number of rows a `SELECT` query that includes a `LIMIT` clause would have returned without that clause.

The news section of the online manual includes a more in-depth list of features. See `http://dev.mysql.com/doc/mysql/en/News.html`.

### 1.5.1.2 The Embedded MySQL Server

The `libmysqld` embedded server library makes MySQL Server suitable for a vastly expanded realm of applications. By using this library, developers can embed MySQL Server into various applications and electronics devices, where the end user has no knowledge of there actually being an underlying database. Embedded MySQL Server is ideal for use behind the scenes in Internet appliances, public kiosks, turnkey hardware/software combination units, high performance Internet servers, self-contained databases distributed on CD-ROM, and so on.

Many users of `libmysqld` will benefit from the MySQL Dual Licensing. For those not wishing to be bound by the GPL, the software is also made available under a commercial license. The embedded MySQL library uses the same interface as the normal client library, so it is convenient and easy to use.

### 1.5.2 MySQL 4.1 in a Nutshell

MySQL Server 4.0 laid the foundation for new features implemented in MySQL 4.1, such as subqueries and Unicode support, and for the work on stored procedures being done in Version 5.0. These features come at the top of the wish list of many of our customers.

With these additions, critics of the MySQL Database Server have to be more imaginative than ever in pointing out deficiencies in the MySQL database management system. Already well-known for its stability, speed, and ease of use, MySQL Server is able to fulfill the requirement checklists of very demanding buyers.

#### 1.5.2.1 Features Available in MySQL 4.1

The MySQL 4.1 features listed in this section already are implemented. A few other MySQL 4.1 features are still planned. See Section 1.6.1, "New Features Planned for 4.1."

The set of features being added to Version 4.1 is mostly fixed. Most new features being coded are or will be available in MySQL 5.0. See Section 1.6.2, "New Features Planned for 5.0."

MySQL 4.1 is currently in the beta stage, and binaries are available for download at `http://dev.mysql.com/downloads/mysql/4.1.html`. All binary releases pass our extensive test suite without any errors on the platforms on which we test.

For those wishing to use the most recent development source for MySQL 4.1, we make our 4.1 BitKeeper repository publicly available.

MySQL 4.1 is going through the steps of *Alpha* (during which time new features might still be added/changed), *Beta* (when we have feature freeze and only bug corrections will be done), and *Gamma* (indicating that a production release is just weeks ahead). At the end of this process, MySQL 4.1 will become the new production release.

- Support for subqueries and derived tables
    - A "subquery" is a `SELECT` statement nested within another statement. A "derived table" (an unnamed view) is a subquery in the `FROM` clause of another statement.
- Speed enhancements
    - Faster binary client/server protocol with support for prepared statements and parameter binding.
    - `BTREE` indexing is now supported for `HEAP` tables, significantly improving response time for non-exact searches.

- New functionality
  - `CREATE TABLE` *`tbl_name2`* `LIKE` *`tbl_name1`* allows you to create, with a single statement, a new table with a structure exactly like that of an existing table.
  - The `MyISAM` storage engine now supports OpenGIS spatial types for storing geographical data.
  - Replication can be done over SSL connections.
- Standards compliance, portability, and migration
  - The new client/server protocol adds the ability to pass multiple warnings to the client, rather than only a single result. This makes it much easier to track problems that occur in operations such as bulk data loading.
  - `SHOW WARNINGS` shows warnings for the last command.
- Internationalization
  - To support applications that require the use of local languages, the MySQL software now offers extensive Unicode support through the `utf8` and `ucs2` character sets.
  - Character sets can now be defined per column, table, and database. This allows for a high degree of flexibility in application design, particularly for multi-language Web sites.
- Usability enhancements
  - In response to popular demand, we have added a server-based `HELP` command that can be used to get help information for SQL statements. The advantage of having this information on the server side is that the information is always applicable to the particular server version that you actually are using. Because this information is available by issuing an SQL statement, any client can be written to access it. For example, the `help` command of the `mysql` command-line client has been modified to have this capability.
  - In the new client/server protocol, multiple statements can be issued with a single call.
  - The new client/server protocol also supports returning multiple result sets. This might occur as a result of sending multiple statements, for example.
  - A new `INSERT ... ON DUPLICATE KEY UPDATE ...` syntax has been implemented. This allows you to `UPDATE` an existing row if the `INSERT` would have caused a duplicate in a `PRIMARY` or `UNIQUE` index.
  - A new aggregate function, `GROUP_CONCAT()`, adds the extremely useful capability of concatenating column values from grouped rows into a single result string.

The news section of the online manual includes a more in-depth list of features. See `http://dev.mysql.com/doc/mysql/en/News.html`.

### 1.5.3 MySQL 5.0: The Next Development Release

New development for MySQL is focused on the 5.0 release, featuring stored procedures and other new features. See Section 1.6.2, "New Features Planned for 5.0."

For those wishing to take a look at the bleeding edge of MySQL development, we make our BitKeeper repository for MySQL Version 5.0 publicly available. As of December 2003, binary builds of Version 5.0 are also available.

## 1.6 MySQL and the Future (the TODO)

This section summarizes the features that we plan to implement in MySQL Server. The items are ordered by release series. Within a list, items are shown in approximately the order they will be done.

**Note:** If you are an enterprise-level user with an urgent need for a particular feature, please contact `sales@mysql.com` to discuss sponsoring options. Targeted financing by sponsor companies allows us to allocate additional resources for specific purposes. One example of a feature sponsored in the past is replication.

### 1.6.1 New Features Planned for 4.1

The following features are not yet implemented in MySQL 4.1, but are planned for implementation as MySQL 4.1 moves into its beta phase. For a list what is already done in MySQL 4.1, see Section 1.5.2.1, "Features Available in MySQL 4.1."

- Stable OpenSSL support (MySQL 4.0 supports rudimentary, not 100% tested, support for OpenSSL).
- More testing of prepared statements.
- More testing of multiple character sets for one table.

### 1.6.2 New Features Planned for 5.0

The following features are planned for inclusion into MySQL 5.0. Some of the features such as stored procedures are complete and are included in MySQL 5.0 alpha, which is available now. Others such as cursors are only partially available. Expect these and other features to mature and be fully supported in upcoming releases.

Note that because we have many developers that are working on different projects, there will also be many additional features. There is also a small chance that some of these features will be added to MySQL 4.1. For a list what is already done in MySQL 4.1, see Section 1.5.2.1, "Features Available in MySQL 4.1."

For those wishing to take a look at the bleeding edge of MySQL development, we make our BitKeeper repository for MySQL Version 5.0 publicly available. As of December 2003, binary builds of Version 5.0 are also available.

- Stored Procedures
  - Stored procedures currently are implemented, based on the SQL:2003 standard.
- New functionality
  - Elementary cursor support.
  - The ability to specify explicitly for `MyISAM` tables that an index should be created as an `RTREE` index. (In MySQL 4.1, `RTREE` indexes are used internally for geometrical data that use GIS data types, but cannot be created on request.)
  - Dynamic length rows for `MEMORY` tables.
- Standards compliance, portability, and migration
  - Add true `VARCHAR` support (column lengths longer than 255, and no stripping of trailing whitespace). There is already support for this in the `MyISAM` storage engine, but it is not yet available at the user level.
- Speed enhancements
  - `SHOW COLUMNS FROM` *`tbl_name`* (used by the `mysql` client to allow expansions of column names) should not open the table, only the definition file. This will require less memory and be much faster.
  - Allow `DELETE` on `MyISAM` tables to use the record cache. To do this, we need to update the threads record cache when we update the `.MYD` file.
  - Better support for `MEMORY` tables:
  - Dynamic length rows.
  - Faster row handling (less copying).
- Usability enhancements
  - Resolving the issue of `RENAME TABLE` on a table used in an active `MERGE` table possibly corrupting the table.

The news section of the online manual includes a more in-depth list of features. See `http://dev.mysql.com/doc/mysql/en/News.html`.

## 1.6.3 New Features Planned for 5.1

- New functionality
  - `FOREIGN KEY` support for all table types, not just `InnoDB`.
  - Column-level constraints.
  - Online backup with very low performance penalty. The online backup will make it easy to add a new replication slave without taking down the master.

- Speed enhancements
    - New text-based table definition file format (`.frm` files) and a table cache for table definitions. This will enable us to do faster queries of table structures and do more efficient foreign key support.
    - Optimize the `BIT` type to take one bit. (`BIT` now takes one byte; it is treated as a synonym for `TINYINT`.)
- Usability enhancements
    - Add options to the client/server protocol to get progress notes for long running commands.
    - Implement `RENAME DATABASE`. To make this safe for all storage engines, it should work as follows:
        1. Create the new database.
        2. For every table, do a rename of the table to another database, as we do with the `RENAME` command.
        3. Drop the old database.
    - New internal file interface change. This will make all file handling much more general and make it easier to add extensions like RAID.

## 1.6.4 New Features Planned for the Near Future

- New functionality
- Views, implemented in stepwise fashion up to full functionality.
    - Oracle-like `CONNECT BY PRIOR` to search tree-like (hierarchical) structures.
    - Add all missing standard SQL and ODBC 3.0 types.
    - Add `SUM(DISTINCT)`.
    - `INSERT SQL_CONCURRENT` and `mysqld --concurrent-insert` to do a concurrent insert at the end of a table if the table is read-locked.
    - Allow variables to be updated in `UPDATE` statements. For example: `UPDATE foo SET @a:=a+b,a=@a, b=@a+c`.
    - Change when user variables are updated so that you can use them with `GROUP BY`, as in the following statement: `SELECT id, @a:=COUNT(*), SUM(sum_col)/@a FROM` *`tbl_name`* `GROUP BY id`.
    - Add an `IMAGE` option to `LOAD DATA INFILE` to not update `TIMESTAMP` and `AUTO_INCREMENT` columns.
    - Add `LOAD DATA INFILE ... UPDATE` syntax that works like this:
        - For tables with primary keys, if an input record contains a primary key value, existing rows matching that primary key value are updated from the remainder of the input columns. However, columns corresponding to columns that are *missing* from the input record are not touched.

        - For tables with primary keys, if an input record does not contain the primary key value or is missing some part of the key, the record is treated as `LOAD DATA INFILE ... REPLACE INTO`.
    - Make `LOAD DATA INFILE` understand syntax like this:

      ```
      LOAD DATA INFILE 'file_name.txt' INTO TABLE tbl_name
           TEXT_FIELDS (text_col1, text_col2, text_col3)
           SET table_col1=CONCAT(text_col1, text_col2),
               table_col3=23
           IGNORE text_col3
      ```

      This can be used to skip over extra columns in the text file, or update columns based on expressions of the read data.
    - New functions for working with `SET` type columns:
        - `ADD_TO_SET(value,set)`
        - `REMOVE_FROM_SET(value,set)`
    - If you abort `mysql` in the middle of a query, you should open another connection and kill the old running query. Alternatively, an attempt should be made to detect this in the server.
    - Add a storage engine interface for table information so that you can use it as a system table. This would be a bit slow if you requested information about all tables, but very flexible. `SHOW INFO FROM` *`tbl_name`* for basic table information should be implemented.
    - Allow `SELECT a FROM` *`tbl_name1`* `LEFT JOIN` *`tbl_name2`* `USING (a)`; in this case `a` is assumed to come from *`tbl_name1`*.
    - `DELETE` and `REPLACE` options to the `UPDATE` statement (this will delete rows when a duplicate-key error occurs while updating).
    - Change the format of `DATETIME` to store fractions of seconds.
    - Make it possible to use the new GNU `regexp` library instead of the current one (the new library should be much faster than the current one).
- Standards compliance, portability, and migration
    - Don't add automatic `DEFAULT` values to columns. Produce an error for any `INSERT` statement that is missing a value for a column that has no `DEFAULT`.
    - Add `ANY()`, `EVERY()`, and `SOME()` group functions. In standard SQL, these work only on boolean columns, but we can extend these to work on any columns or expressions by treating a value of zero as FALSE and non-zero values as TRUE.
    - Fix the type of `MAX(column)` to be the same as the column type:

      ```
      mysql> CREATE TABLE t1 (a DATE);
      mysql> INSERT INTO t1 VALUES (NOW());
      mysql> CREATE TABLE t2 SELECT MAX(a) FROM t1;
      mysql> SHOW COLUMNS FROM t2;
      ```

- Speed enhancements
  - Don't allow more than a defined number of threads to run `MyISAM` recovery at the same time.
  - Change `INSERT INTO ... SELECT` to optionally use concurrent inserts.
  - Add an option to periodically flush key pages for tables with delayed keys if they haven't been used in a while.
  - Allow join on key parts (optimization issue).
  - Add a log file analyzer that can extract information about which tables are hit most often, how often multiple-table joins are executed, and so on. This should help users identify areas of table design that could be optimized to execute much more efficient queries.
- Usability enhancements
  - Return the original column types when doing `SELECT MIN(`*`column`*`) ... GROUP BY`.
  - Make it possible to specify `long_query_time` with a granularity in microseconds.
  - Link the `myisampack` code into the server so that it can perform `PACK` or `COMPRESS` operations.
  - Add a temporary key buffer cache during `INSERT/DELETE/UPDATE` so that we can gracefully recover if the index file gets full.
  - If you perform an `ALTER TABLE` on a table that is symlinked to another disk, create temporary tables on that disk.
  - Implement a `DATE/DATETIME` type that handles time zone information properly, to make dealing with dates in different time zones easier.
  - Fix `configure` so that all libraries (like `MyISAM`) can be compiled without threads.
  - Allow user variables as `LIMIT` arguments; for example, `LIMIT @a,@b`.
  - Automatic output from `mysql` to a Web browser.
  - `LOCK DATABASES` (with various options).
  - Many more variables for `SHOW STATUS`. Record reads and updates. Selects on a single table and selects with joins. Mean number of tables in selects. Number of `ORDER BY` and `GROUP BY` queries.
  - `mysqladmin copy` *`database new-database`*; this requires a `COPY` operation to be added to `mysqld`.
  - Processlist output should indicate the number of queries/threads.
  - `SHOW HOSTS` for printing information about the hostname cache.
  - Change table names from empty strings to `NULL` for calculated columns.
  - Don't use `Item_copy_string` on numerical values to avoid number-to-string-to-number conversion in case of `SELECT COUNT(*)*(id+0) FROM` *`tbl_name`* `GROUP BY id`.

    - Change so that `ALTER TABLE` doesn't abort clients that execute `INSERT DELAYED`.
    - Fix so that when columns are referenced in an `UPDATE` clause, they contain the old values from before the update started.
- New operating systems
    - Port the MySQL clients to LynxOS.

### 1.6.5 New Features Planned for the Mid-Term Future

- Implement function: `get_changed_tables(`*`timeout`*`,`*`table1`*`,`*`table2`*`,...)`.
- Change reading through tables to use `mmap()` when possible. Now only compressed tables use `mmap()`.
- Make the automatic timestamp code nicer. Add timestamps to the update log with `SET TIMESTAMP=`*`val`*`;`.
- Use read/write mutex in some places to get more speed.
- Automatically close some tables if a table, temporary table, or temporary file gets error 23 (too many open files).
- Better constant propagation. When an occurrence of *`col_name`*`=`*`n`* is found in an expression, for some constant *`n`*, replace other occurrences of *`col_name`* within the expression with *`n`*. Currently, this is done only for some simple cases.
- Change all const expressions with calculated expressions if possible.
- Optimize *`key`* `=` *`expr`* comparisons. At the moment, only *`key`* `=` *`column`* or *`key`* `=` *`constant`* comparisons are optimized.
- Join some of the copy functions for nicer code.
- Change `sql_yacc.yy` to an inline parser to reduce its size and get better error messages.
- Change the parser to use only one rule per different number of arguments in function.
- Use of full calculation names in the order part (for Access97).
- `MINUS`, `INTERSECT`, and `FULL OUTER JOIN`. (Currently `UNION` and `LEFT|RIGHT OUTER JOIN` are supported.)
- Allow `SQL_OPTION MAX_SELECT_TIME=`*`val`*, for placing a time limit on a query.
- Allow updates to be logged to a database.
- Enhance `LIMIT` to allow retrieval of data from the end of a result set.
- Alarm around client connect/read/write functions.
- Please note the changes to `mysqld_safe`: According to FSSTND (which Debian tries to follow), PID files should go into `/var/run/<progname>.pid` and log files into `/var/log`. It would be nice if you could put the "DATADIR" in the first declaration of "pidfile" and "log" so that the placement of these files can be changed with a single statement.

- Allow a client to request logging.
- Allow the `LOAD DATA INFILE` statement to read files that have been compressed with `gzip`.
- Fix sorting and grouping of `BLOB` columns (partly solved now).
- Change to use semaphores when counting threads. One should first implement a semaphore library for MIT-pthreads.
- Add full support for `JOIN` with parentheses.
- As an alternative to the one-thread-per-connection model, manage a pool of threads to handle queries.
- Allow `GET_LOCK()` to obtain more than one lock. When doing this, it is also necessary to handle the possible deadlocks this change will introduce.

### 1.6.6 New Features We Don't Plan to Implement

We aim toward full compliance with ANSI/ISO SQL. There are no features we plan not to implement.

## 1.7 MySQL Information Sources

### 1.7.1 MySQL Mailing Lists

This section introduces the MySQL mailing lists and provides guidelines as to how the lists should be used. When you subscribe to a mailing list, you will receive all postings to the list as email messages. You can also send your own questions and answers to the list.

#### 1.7.1.1 The MySQL Mailing Lists

To subscribe to or unsubscribe from any of the mailing lists described in this section, visit `http://lists.mysql.com/`. Please *do not* send messages about subscribing or unsubscribing to any of the mailing lists, because such messages are distributed automatically to thousands of other users.

Your local site may have many subscribers to a MySQL mailing list. If so, the site may have a local mailing list, so that messages sent from `lists.mysql.com` to your site are propagated to the local list. In such cases, please contact your system administrator to be added to or dropped from the local MySQL list.

If you wish to have traffic for a mailing list go to a separate mailbox in your mail program, set up a filter based on the message headers. You can use either the `List-ID:` or `Delivered-To:` headers to identify list messages.

The MySQL mailing lists are as follows:

- `announce`

  This list is for announcements of new versions of MySQL and related programs. This is a low-volume list to which all MySQL users should subscribe.
- `mysql`

  This is the main list for general MySQL discussion. Please note that some topics are better discussed on the more-specialized lists. If you post to the wrong list, you may not get an answer.
- `mysql-digest`

  This is the `mysql` list in digest form. Subscribing to this list means that you will get all list messages, sent as one large mail message once a day.
- `bugs`

  This list will be of interest to you if you want to stay informed about issues reported since the last release of MySQL or if you want to be actively involved in the process of bug hunting and fixing. See Section 1.7.1.3, "How to Report Bugs or Problems."
- `bugs-digest`

  This is the `bugs` list in digest form.
- `internals`

  This list is for people who work on the MySQL code. This is also the forum for discussions on MySQL development and for posting patches.
- `internals-digest`

  This is the `internals` list in digest form.
- `mysqldoc`

  This list is for people who work on the MySQL documentation: people from MySQL AB, translators, and other community members.
- `mysqldoc-digest`

  This is the `mysqldoc` list in digest form.
- `benchmarks`

  This list is for anyone interested in performance issues. Discussions concentrate on database performance (not limited to MySQL), but also include broader categories such as performance of the kernel, filesystem, disk system, and so on.
- `benchmarks-digest`

  This is the `benchmarks` list in digest form.

- `packagers`

  This list is for discussions on packaging and distributing MySQL. This is the forum used by distribution maintainers to exchange ideas on packaging MySQL and on ensuring that MySQL looks and feels as similar as possible on all supported platforms and operating systems.

- `packagers-digest`

  This is the `packagers` list in digest form.

- `java`

  This list is for discussions about the MySQL server and Java. It is mostly used to discuss JDBC drivers, including MySQL Connector/J.

- `java-digest`

  This is the `java` list in digest form.

- `win32`

  This list is for all topics concerning the MySQL software on Microsoft operating systems, such as Windows 9x, Me, NT, 2000, and XP.

- `win32-digest`

  This is the `win32` list in digest form.

- `myodbc`

  This list is for all topics concerning connecting to the MySQL server with ODBC.

- `myodbc-digest`

  This is the `myodbc` list in digest form.

- `gui-tools`

  This list is for all topics concerning MySQL GUI tools, including `MySQL Administrator` and the `Control Center` graphical client.

- `gui-tools-digest`

  This is the `mysqlcc` list in digest form.

- `plusplus`

  This list is for all topics concerning programming with the C++ API for MySQL.

- `plusplus-digest`

  This is the `plusplus` list in digest form.

- `msql-mysql-modules`

  This list is for all topics concerning the Perl support for MySQL with `msql-mysql-modules`, which is now named `DBD::mysql`.

- `msql-mysql-modules-digest`

  This is the `msql-mysql-modules` list in digest form.

If you're unable to get an answer to your questions from a MySQL mailing list, one option is to purchase support from MySQL AB. This will put you in direct contact with MySQL developers. See Section 1.4.1, "Support Offered by MySQL AB."

The following table shows some MySQL mailing lists in languages other than English. These lists are not operated by MySQL AB.

- `mysql-france-subscribe@yahoogroups.com`

  A French mailing list.
- `list@tinc.net`

  A Korean mailing list. Email `subscribe mysql your@email.address` to this list.
- `mysql-de-request@lists.4t2.com`

  A German mailing list. Email `subscribe mysql-de your@email.address` to this list. You can find information about this mailing list at `http://www.4t2.com/mysql/`.
- `mysql-br-request@listas.linkway.com.br`

  A Portuguese mailing list. Email `subscribe mysql-br your@email.address` to this list.
- `mysql-alta@elistas.net`

  A Spanish mailing list. Email `subscribe mysql your@email.address` to this list.

### 1.7.1.2 Asking Questions or Reporting Bugs

Before posting a bug report or question, please do the following:

- Start by searching the MySQL online manual at `http://dev.mysql.com/doc/`. We try to keep the manual up to date by updating it frequently with solutions to newly found problems. The change history (`http://dev.mysql.com/doc/mysql/en/News.html`) can be particularly useful since it is quite possible that a newer version already contains a solution to your problem.
- Search in the bugs database at `http://bugs.mysql.com/` to see whether the bug has already been reported and fixed.
- Search the MySQL mailing list archives at `http://lists.mysql.com/`.
- You can also use `http://www.mysql.com/search/` to search all the Web pages (including the manual) that are located at the MySQL AB Web site.

If you can't find an answer in the manual or the archives, check with your local MySQL expert. If you still can't find an answer to your question, please follow the guidelines on sending mail to a MySQL mailing list, outlined in the next section, before contacting us.

### 1.7.1.3 How to Report Bugs or Problems

The normal place to report bugs is `http://bugs.mysql.com/`, which is the address for our bugs database. This database is public, and can be browsed and searched by anyone. If you log in to the system, you will also be able to enter new reports.

Writing a good bug report takes patience, but doing it right the first time saves time both for us and for yourself. A good bug report, containing a full test case for the bug, makes it very likely that we will fix the bug in the next release. This section will help you write your report correctly so that you don't waste your time doing things that may not help us much or at all.

We encourage everyone to use the `mysqlbug` script to generate a bug report (or a report about any problem). `mysqlbug` can be found in the `scripts` directory (source distribution) and in the `bin` directory under your MySQL installation directory (binary distribution). If you are unable to use `mysqlbug` (for example, if you are running on Windows), it is still vital that you include all the necessary information noted in this section (most importantly, a description of the operating system and the MySQL version).

The `mysqlbug` script helps you generate a report by determining much of the following information automatically, but if something important is missing, please include it with your message. Please read this section carefully and make sure that all the information described here is included in your report.

Preferably, you should test the problem using the latest production or development version of MySQL Server before posting. Anyone should be able to repeat the bug by just using `mysql test <` *`script_file`* on the included test case or by running the shell or Perl script that is included in the bug report.

All bugs posted in the bugs database at `http://bugs.mysql.com/` will be corrected or documented in the next MySQL release. If only minor code changes are needed to correct a problem, we may also post a patch that fixes the problem.

If you have found a sensitive security bug in MySQL, you can send email to `security@mysql.com`.

If you have a repeatable bug report, please report it to the bugs database at `http://bugs.mysql.com/`. Note that even in this case it's good to run the `mysqlbug` script first to find information about your system. Any bug that we are able to repeat has a high chance of being fixed in the next MySQL release.

To report other problems, you can use one of the MySQL mailing lists.

Remember that it is possible for us to respond to a message containing too much information, but not to one containing too little. People often omit facts because they think they know the cause of a problem and assume that some details don't matter. A good principle is this: If you are in doubt about stating something, state it. It is faster and less troublesome to write a couple more lines in your report than to wait longer for the answer if we must ask you to provide information that was missing from the initial report.

The most common errors made in bug reports are (a) not including the version number of the MySQL distribution used, and (b) not fully describing the platform on which the MySQL server is installed (including the platform type and version number). This is highly relevant information, and in 99 cases out of 100, the bug report is useless without it. Very often we get questions like, "Why doesn't this work for me?" Then we find that the feature requested wasn't implemented in that MySQL version, or that a bug described in a report has already been fixed in newer MySQL versions. Sometimes the error is platform-dependent; in such cases, it is next to impossible for us to fix anything without knowing the operating system and the version number of the platform.

If you compiled MySQL from source, remember also to provide information about your compiler, if it is related to the problem. Often people find bugs in compilers and think the problem is MySQL-related. Most compilers are under development all the time and become better version by version. To determine whether your problem depends on your compiler, we need to know what compiler you use. Note that every compiling problem should be regarded as a bug and reported accordingly.

It is most helpful when a good description of the problem is included in the bug report. That is, give a good example of everything you did that led to the problem and describe, in exact detail, the problem itself. The best reports are those that include a full example showing how to reproduce the bug or problem.

If a program produces an error message, it is very important to include the message in your report. If we try to search for something from the archives using programs, it is better that the error message reported exactly matches the one that the program produces. (Even the lettercase should be observed.) You should never try to reproduce from memory what the error message was; instead, copy and paste the entire message into your report.

If you have a problem with Connector/ODBC (MyODBC), please try to generate a MyODBC trace file and send it with your report.

Please remember that many of the people who will read your report will do so using an 80-column display. When generating reports or examples using the `mysql` command-line tool, you should therefore use the `--vertical` option (or the `\G` statement terminator) for output that would exceed the available width for such a display (for example, with the `EXPLAIN SELECT` statement; see the example later in this section).

Please include the following information in your report:

- The version number of the MySQL distribution you are using (for example, MySQL 4.0.12). You can find out which version you are running by executing `mysqladmin version`. The `mysqladmin` program can be found in the `bin` directory under your MySQL installation directory.
- The manufacturer and model of the machine on which you experience the problem.
- The operating system name and version. If you work with Windows, you can usually get the name and version number by double-clicking your My Computer icon and pulling down the Help/About Windows menu. For most Unix-like operating systems, you can get this information by executing the command `uname -a`.
- Sometimes the amount of memory (real and virtual) is relevant. If in doubt, include these values.

- If you are using a source distribution of the MySQL software, the name and version number of the compiler used are needed. If you have a binary distribution, the distribution name is needed.
- If the problem occurs during compilation, include the exact error messages and also a few lines of context around the offending code in the file where the error occurs.
- If `mysqld` died, you should also report the query that crashed `mysqld`. You can usually find this out by running `mysqld` with query logging enabled, and then looking in the log after `mysqld` crashes.
- If a database table is related to the problem, include the output from `mysqldump --no-data` *`db_name tbl_name`*. This is very easy to do and is a powerful way to get information about any table in a database. The information will help us create a situation matching the one you have.
- For speed-related bugs or problems with `SELECT` statements, you should always include the output of `EXPLAIN SELECT ...`, and at least the number of rows that the `SELECT` statement produces. You should also include the output from `SHOW CREATE TABLE` *`tbl_name`* for each involved table. The more information you give about your situation, the more likely it is that someone can help you.

  The following is an example of a very good bug report. It should be posted with the `mysqlbug` script. The example uses the `mysql` command-line tool. Note the use of the `\G` statement terminator for statements whose output width would otherwise exceed that of an 80-column display device.

  ```
  mysql> SHOW VARIABLES;
  mysql> SHOW COLUMNS FROM ...\G
         <output from SHOW COLUMNS>
  mysql> EXPLAIN SELECT ...\G
         <output from EXPLAIN>
  mysql> FLUSH STATUS;
  mysql> SELECT ...;
         <A short version of the output from SELECT,
         including the time taken to run the query>
  mysql> SHOW STATUS;
         <output from SHOW STATUS>
  ```

- If a bug or problem occurs while running `mysqld`, try to provide an input script that will reproduce the anomaly. This script should include any necessary source files. The more closely the script can reproduce your situation, the better. If you can make a reproducible test case, you should post it on `http://bugs.mysql.com/` for high-priority treatment.

  If you can't provide a script, you should at least include the output from `mysqladmin variables extended-status processlist` in your mail to provide some information on how your system is performing.

- If you can't produce a test case with only a few rows, or if the test table is too big to be mailed to the mailing list (more than 10 rows), you should dump your tables using `mysqldump` and create a `README` file that describes your problem.

  Create a compressed archive of your files using `tar` and `gzip` or `zip`, and use FTP to transfer the archive to `ftp://ftp.mysql.com/pub/mysql/upload/`. Then enter the problem into our bugs database at `http://bugs.mysql.com/`.
- If you think that the MySQL server produces a strange result from a query, include not only the result, but also your opinion of what the result should be, and an account describing the basis for your opinion.
- When giving an example of the problem, it's better to use the variable names, table names, and so on that exist in your actual situation than to come up with new names. The problem could be related to the name of a variable or table. These cases are rare, perhaps, but it is better to be safe than sorry. After all, it should be easier for you to provide an example that uses your actual situation, and it is by all means better for us. In case you have data that you don't want to show to others, you can use FTP to transfer it to `ftp://ftp.mysql.com/pub/mysql/upload/`. If the information is really top secret and you don't want to show it even to us, then go ahead and provide an example using other names, but please regard this as the last choice.
- Include all the options given to the relevant programs, if possible. For example, indicate the options that you use when you start the `mysqld` server as well as the options that you use to run any MySQL client programs. The options to programs such as `mysqld` and `mysql`, and to the `configure` script, are often keys to answers and are very relevant. It is never a bad idea to include them. If you use any modules, such as Perl or PHP, please include the version numbers of those as well.
- If your question is related to the privilege system, please include the output of `mysqlaccess`, the output of `mysqladmin reload`, and all the error messages you get when trying to connect. When you test your privileges, you should first run `mysqlaccess`. After this, execute `mysqladmin reload version` and try to connect with the program that gives you trouble. `mysqlaccess` can be found in the `bin` directory under your MySQL installation directory.
- If you have a patch for a bug, do include it. But don't assume that the patch is all we need, or that we will use it, if you don't provide some necessary information such as test cases showing the bug that your patch fixes. We might find problems with your patch or we might not understand it at all; if so, we can't use it.

  If we can't verify exactly what the purpose of the patch is, we won't use it. Test cases will help us here. Show that the patch will handle all the situations that may occur. If we find a borderline case (even a rare one) where the patch won't work, it may be useless.
- Guesses about what the bug is, why it occurs, or what it depends on are usually wrong. Even the MySQL team can't guess such things without first using a debugger to determine the real cause of a bug.

- Indicate in your bug report that you have checked the reference manual and mail archive so that others know you have tried to solve the problem yourself.
- If you get a `parse error`, please check your syntax closely. If you can't find something wrong with it, it's extremely likely that your current version of MySQL Server doesn't support the syntax you are using. If you are using the current version and the manual at `http://dev.mysql.com/doc/` doesn't cover the syntax you are using, MySQL Server doesn't support your query. In this case, your only options are to implement the syntax yourself or email `licensing@mysql.com` and ask for an offer to implement it.

  If the manual covers the syntax you are using, but you have an older version of MySQL Server, you should check the MySQL change history to see when the syntax was implemented. In this case, you have the option of upgrading to a newer version of MySQL Server.
- If your problem is that your data appears corrupt or you get errors when you access a particular table, you should first check and then try to repair your tables with `CHECK TABLE` and `REPAIR TABLE` or with `myisamchk`.

  If you are running Windows, please verify that `lower_case_table_names` is 1 or 2 with `SHOW VARIABLES LIKE 'lower_case_table_names'`.
- If you often get corrupted tables, you should try to find out when and why this happens. In this case, the error log in the MySQL data directory may contain some information about what happened. (This is the file with the `.err` suffix in the name.) Please include any relevant information from this file in your bug report. Normally `mysqld` should *never* crash a table if nothing killed it in the middle of an update. If you can find the cause of `mysqld` dying, it's much easier for us to provide you with a fix for the problem.
- If possible, download and install the most recent version of MySQL Server and check whether it solves your problem. All versions of the MySQL software are thoroughly tested and should work without problems. We believe in making everything as backward-compatible as possible, and you should be able to switch MySQL versions without difficulty.

If you are a support customer, please cross-post the bug report to `mysql-support@mysql.com` for higher-priority treatment, as well as to the appropriate mailing list to see whether someone else has experienced (and perhaps solved) the problem.

When answers are sent to you individually and not to the mailing list, it is considered good etiquette to summarize the answers and send the summary to the mailing list so that others may have the benefit of responses you received that helped you solve your problem.

### 1.7.1.4 Guidelines for Answering Questions on the Mailing List

If you consider your answer to have broad interest, you may want to post it to the mailing list instead of replying directly to the individual who asked. Try to make your answer general enough that people other than the original poster may benefit from it. When you post to the list, please make sure that your answer is not a duplication of a previous answer.

Try to summarize the essential part of the question in your reply; don't feel obliged to quote the entire original message.

Please don't post mail messages from your browser with HTML mode turned on. Many users don't read mail with a browser.

## 1.7.2 MySQL Community Support on IRC (Internet Relay Chat)

In addition to the various MySQL mailing lists, you can find experienced community people on `IRC` (`Internet Relay Chat`). These are the best networks/channels currently known to us:

- **freenode** (see `http://www.freenode.net/` for servers)
  - `#mysql` Primarily MySQL questions, but other database and SQL questions are welcome.
  - `#mysqlphp` Questions about MySQL+PHP, a popular combination.
  - `#mysqlperl` Questions about MySQL+Perl, another popular combination.
- **EFnet** (see `http://www.efnet.org/` for servers)
  - `#mysql` MySQL questions.

If you are looking for IRC client software to connect to an IRC network, take a look at `X-Chat` (`http://www.xchat.org/`). X-Chat (GPL licensed) is available for Unix as well as for Windows platforms.

## 1.8 MySQL Standards Compliance

This section describes how MySQL relates to the ANSI/ISO SQL standards. MySQL Server has many extensions to the SQL standard, and here you will find out what they are and how to use them. You will also find information about functionality missing from MySQL Server, and how to work around some differences.

The SQL standard has been evolving since 1986 and several versions exist. In this manual, "SQL-92" refers to the standard released in 1992, "SQL:1999" refers to the standard released in 1999, and "SQL:2003" refers to the current version of the standard. We use the phrase "the SQL standard" to mean the current version of the SQL Standard at any time.

Our goal is to not restrict MySQL Server usability for any usage without a very good reason for doing so. Even if we don't have the resources to perform development for every possible use, we are always willing to help and offer suggestions to people who are trying to use MySQL Server in new territories.

One of our main goals with the product is to continue to work toward compliance with the SQL standard, but without sacrificing speed or reliability. We are not afraid to add extensions to SQL or support for non-SQL features if this greatly increases the usability of MySQL Server for a large segment of our user base. The `HANDLER` interface in MySQL Server 4.0 is an example of this strategy.

We will continue to support transactional and non-transactional databases to satisfy both mission-critical 24/7 usage and heavy Web or logging usage.

MySQL Server was originally designed to work with medium size databases (10-100 million rows, or about 100MB per table) on small computer systems. Today MySQL Server handles terabyte-size databases, but the code can also be compiled in a reduced version suitable for hand-held and embedded devices. The compact design of the MySQL server makes development in both directions possible without any conflicts in the source tree.

Currently, we are not targeting realtime support, although MySQL replication capabilities already offer significant functionality.

Database cluster support is planned through integration of our acquired NDB Cluster technology into a new storage engine, available in 2004.

We are also looking at providing XML support in the database server.

## 1.8.1 What Standards MySQL Follows

We are aiming toward supporting the full ANSI/ISO SQL standard, but without making concessions to speed and quality of the code.

ODBC levels 0–3.51.

## 1.8.2 Selecting SQL Modes

The MySQL server can operate in different SQL modes, and can apply these modes differentially for different clients. This allows applications to tailor server operation to their own requirements.

Modes define what SQL syntax MySQL should support and what kind of validation checks it should perform on the data. This makes it easier to use MySQL in a lot of different environments and to use MySQL together with other database servers.

You can set the default SQL mode by starting `mysqld` with the `--sql-mode="modes"` option. Beginning with MySQL 4.1, you can also change the mode after startup time by setting the `sql_mode` variable with a `SET [SESSION|GLOBAL] sql_mode='modes'` statement.

## 1.8.3 Running MySQL in ANSI Mode

You can tell `mysqld` to use the ANSI mode with the `--ansi` startup option.

Running the server in ANSI mode is the same as starting it with these options (specify the `--sql_mode` value on a single line):

```
--transaction-isolation=SERIALIZABLE
--sql-mode=REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,
IGNORE_SPACE,ONLY_FULL_GROUP_BY
```

In MySQL 4.1, you can achieve the same effect with these two statements (specify the `sql_mode` value on a single line):

```
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET GLOBAL sql_mode = 'REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,
IGNORE_SPACE,ONLY_FULL_GROUP_BY';
```

See Section 1.8.2, "Selecting SQL Modes."

In MySQL 4.1.1, the `sql_mode` options shown can be also be set with this statement:

```
SET GLOBAL sql_mode='ansi';
```

In this case, the value of the `sql_mode` variable will be set to all options that are relevant for ANSI mode. You can check the result like this:

```
mysql> SET GLOBAL sql_mode='ansi';
mysql> SELECT @@global.sql_mode;
        -> 'REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,
              IGNORE_SPACE,ONLY_FULL_GROUP_BY,ANSI';
```

## 1.8.4 MySQL Extensions to Standard SQL

MySQL Server includes some extensions that you probably will not find in other SQL databases. Be warned that if you use them, your code will not be portable to other SQL servers. In some cases, you can write code that includes MySQL extensions, but is still portable, by using comments of the form `/*! ... */`. In this case, MySQL Server will parse and execute the code within the comment as it would any other MySQL statement, but other SQL servers will ignore the extensions. For example:

```
SELECT /*! STRAIGHT_JOIN */ col_name FROM table1,table2 WHERE ...
```

If you add a version number after the '`!`' character, the syntax within the comment will be executed only if the MySQL version is equal to or newer than the specified version number:

```
CREATE /*!32302 TEMPORARY */ TABLE t (a INT);
```

This means that if you have Version 3.23.02 or newer, MySQL Server will use the `TEMPORARY` keyword.

The following descriptions list MySQL extensions, organized by category.

- Organization of data on disk

  MySQL Server maps each database to a directory under the MySQL data directory, and tables within a database to filenames in the database directory. This has a few implications:

  - Database names and table names are case sensitive in MySQL Server on operating systems that have case-sensitive filenames (such as most Unix systems).

    - You can use standard system commands to back up, rename, move, delete, and copy tables that are managed by the `MyISAM` or `ISAM` storage engines. For example, to rename a `MyISAM` table, rename the `.MYD`, `.MYI`, and `.frm` files to which the table corresponds.

  Database, table, index, column, or alias names may begin with a digit (but may not consist solely of digits).

- General language syntax
    - Strings may be enclosed by either ‘“’ or ‘’’, not just by ‘’’.
    - Use of ‘\’ as an escape character in strings.
    - In SQL statements, you can access tables from different databases with the *`db_name.tbl_name`* syntax. Some SQL servers provide the same functionality but call this `User space`. MySQL Server doesn’t support tablespaces such as used in statements like this: `CREATE TABLE ralph.my_table...IN my_tablespace`.
- SQL statement syntax
    - The `ANALYZE TABLE`, `CHECK TABLE`, `OPTIMIZE TABLE`, and `REPAIR TABLE` statements.
    - The `CREATE DATABASE` and `DROP DATABASE` statements.
    - The `DO` statement.
    - `EXPLAIN SELECT` to get a description of how tables are joined.
    - The `FLUSH` and `RESET` statements.
    - The `SET` statement.
    - The `SHOW` statement.
    - Use of `LOAD DATA INFILE`. In many cases, this syntax is compatible with Oracle’s `LOAD DATA INFILE`.
    - Use of `RENAME TABLE`.
    - Use of `REPLACE` instead of `DELETE` + `INSERT`.
    - Use of `CHANGE` *`col_name`*, `DROP` *`col_name`*, or `DROP INDEX`, `IGNORE` or `RENAME` in an `ALTER TABLE` statement. Use of multiple `ADD`, `ALTER`, `DROP`, or `CHANGE` clauses in an `ALTER TABLE` statement.
    - Use of index names, indexes on a prefix of a field, and use of `INDEX` or `KEY` in a `CREATE TABLE` statement.
    - Use of `TEMPORARY` or `IF NOT EXISTS` with `CREATE TABLE`.
    - Use of `IF EXISTS` with `DROP TABLE`.
    - You can drop multiple tables with a single `DROP TABLE` statement.
    - The `ORDER BY` and `LIMIT` clauses of the `UPDATE` and `DELETE` statements.
    - `INSERT INTO ... SET` *`col_name`* `= ...` syntax.
    - The `DELAYED` clause of the `INSERT` and `REPLACE` statements.

  - The `LOW_PRIORITY` clause of the `INSERT`, `REPLACE`, `DELETE`, and `UPDATE` statements.
  - Use of `INTO OUTFILE` and `STRAIGHT_JOIN` in a `SELECT` statement.
  - The `SQL_SMALL_RESULT` option in a `SELECT` statement.
  - You don't need to name all selected columns in the `GROUP BY` part. This gives better performance for some very specific, but quite normal queries.
  - You can specify `ASC` and `DESC` with `GROUP BY`.
  - The ability to set variables in a statement with the `:=` assignment operator:

    ```
    mysql> SELECT @a:=SUM(total),@b=COUNT(*),@a/@b AS avg
        -> FROM test_table;
    mysql> SELECT @t1:=(@t2:=1)+@t3:=4,@t1,@t2,@t3;
    ```

- Column types
  - The column types `MEDIUMINT`, `SET`, `ENUM`, and the different `BLOB` and `TEXT` types.
  - The column attributes `AUTO_INCREMENT`, `BINARY`, `NULL`, `UNSIGNED`, and `ZEROFILL`.
- Functions and operators
  - To make it easier for users who come from other SQL environments, MySQL Server supports aliases for many functions. For example, all string functions support both standard SQL syntax and ODBC syntax.
  - MySQL Server understands the `||` and `&&` operators to mean logical OR and AND, as in the C programming language. In MySQL Server, `||` and `OR` are synonyms, as are `&&` and `AND`. Because of this nice syntax, MySQL Server doesn't support the standard SQL `||` operator for string concatenation; use `CONCAT()` instead. Because `CONCAT()` takes any number of arguments, it's easy to convert use of the `||` operator to MySQL Server.
  - Use of `COUNT(DISTINCT` *`list`*`)` where *`list`* has more than one element.
  - All string comparisons are case-insensitive by default, with sort ordering determined by the current character set (ISO-8859-1 Latin1 by default). If you don't like this, you should declare your columns with the `BINARY` attribute or use the `BINARY` cast, which causes comparisons to be done using the underlying character code values rather then a lexical ordering.
  - The `%` operator is a synonym for `MOD()`. That is, *`N`* `%` *`M`* is equivalent to `MOD(`*`N`*`,`*`M`*`)`. `%` is supported for C programmers and for compatibility with PostgreSQL.
  - The `=`, `<>`, `<=`,`<`, `>=`,`>`, `<<`, `>>`, `<=>`, `AND`, `OR`, or `LIKE` operators may be used in column comparisons to the left of the `FROM` in `SELECT` statements. For example:

    ```
    mysql> SELECT col1=1 AND col2=2 FROM tbl_name;
    ```

  - The `LAST_INSERT_ID()` function that returns the most recent `AUTO_INCREMENT` value.
  - `LIKE` is allowed on numeric columns.
  - The `REGEXP` and `NOT REGEXP` extended regular expression operators.

- `CONCAT()` or `CHAR()` with one argument or more than two arguments. (In MySQL Server, these functions can take any number of arguments.)
- The `BIT_COUNT()`, `CASE`, `ELT()`, `FROM_DAYS()`, `FORMAT()`, `IF()`, `PASSWORD()`, `ENCRYPT()`, `MD5()`, `ENCODE()`, `DECODE()`, `PERIOD_ADD()`, `PERIOD_DIFF()`, `TO_DAYS()`, and `WEEKDAY()` functions.
- Use of `TRIM()` to trim substrings. Standard SQL supports removal of single characters only.
- The `GROUP BY` functions `STD()`, `BIT_OR()`, `BIT_AND()`, `BIT_XOR()`, and `GROUP_CONCAT()`.

For a prioritized list indicating when new extensions will be added to MySQL Server, you should consult the online MySQL TODO list at `http://dev.mysql.com/doc/mysql/en/TODO.html`. That is the latest version of the TODO list in this manual. See Section 1.6, "MySQL and the Future (the TODO)."

## 1.8.5 MySQL Differences from Standard SQL

We try to make MySQL Server follow the ANSI SQL standard and the ODBC SQL standard, but MySQL Server performs operations differently in some cases:

- For `VARCHAR` columns, trailing spaces are removed when the value is stored. See Section 1.8.7, "Known Errors and Design Deficiencies in MySQL."
- In some cases, `CHAR` columns are silently converted to `VARCHAR` columns when you define a table or alter its structure.
- Privileges for a table are not automatically revoked when you delete a table. You must explicitly issue a `REVOKE` statement to revoke privileges for a table.

### 1.8.5.1 Subqueries

MySQL 4.1 supports subqueries and derived tables. A "subquery" is a `SELECT` statement nested within another statement. A "derived table" (an unnamed view) is a subquery in the `FROM` clause of another statement.

For MySQL versions older than 4.1, most subqueries can be rewritten using joins or other methods.

### 1.8.5.2 `SELECT INTO TABLE`

MySQL Server doesn't support the Sybase SQL extension: `SELECT ... INTO TABLE ....` Instead, MySQL Server supports the standard SQL syntax `INSERT INTO ... SELECT ...`, which is basically the same thing.

```
INSERT INTO tbl_temp2 (fld_id)
    SELECT tbl_temp1.fld_order_id
    FROM tbl_temp1 WHERE tbl_temp1.fld_order_id > 100;
```

Alternatively, you can use `SELECT INTO OUTFILE ...` or `CREATE TABLE ... SELECT`.

From Version 5.0, MySQL supports `SELECT ... INTO` with user variables. The same syntax may also be used inside stored procedures using cursors and local variables.

### 1.8.5.3 Transactions and Atomic Operations

MySQL Server (Version 3.23-max and all Versions 4.0 and above) supports transactions with the `InnoDB` and `BDB` transactional storage engines. `InnoDB` provides *full* `ACID` compliance.

The other non-transactional storage engines in MySQL Server (such as `MyISAM`) follow a different paradigm for data integrity called "atomic operations." In transactional terms, `MyISAM` tables effectively always operate in `AUTOCOMMIT=1` mode. Atomic operations often offer comparable integrity with higher performance.

With MySQL Server supporting both paradigms, you can decide whether your applications are best served by the speed of atomic operations or the use of transactional features. This choice can be made on a per-table basis.

As noted, the trade-off for transactional versus non-transactional table types lies mostly in performance. Transactional tables have significantly higher memory and diskspace requirements, and more CPU overhead. On the other hand, transactional table types such as `InnoDB` also offer many significant features. MySQL Server's modular design allows the concurrent use of different storage engines to suit different requirements and deliver optimum performance in all situations.

But how do you use the features of MySQL Server to maintain rigorous integrity even with the non-transactional `MyISAM` tables, and how do these features compare with the transactional table types?

1. If your applications are written in a way that is dependent on being able to call `ROLLBACK` rather than `COMMIT` in critical situations, transactions are more convenient. Transactions also ensure that unfinished updates or corrupting activities are not committed to the database; the server is given the opportunity to do an automatic rollback and your database is saved.

   If you use non-transactional tables, MySQL Server in almost all cases allows you to resolve potential problems by including simple checks before updates and by running simple scripts that check the databases for inconsistencies and automatically repair or warn if such an inconsistency occurs. Note that just by using the MySQL log or even adding one extra log, you can normally fix tables perfectly with no data integrity loss.
2. More often than not, critical transactional updates can be rewritten to be atomic. Generally speaking, all integrity problems that transactions solve can be done with `LOCK TABLES` or atomic updates, ensuring that you never will get an automatic abort from the server, which is a common problem with transactional database systems.
3. Even a transactional system can lose data if the server goes down. The difference between different systems lies in just how small the time-lag is where they could lose data. No

system is 100% secure, only “secure enough.” Even Oracle, reputed to be the safest of transactional database systems, is reported to sometimes lose data in such situations.

To be safe with MySQL Server, whether or not using transactional tables, you only need to have backups and have binary logging turned on. With this, you can recover from any situation that you could with any other transactional database system. It is always good to have backups, regardless of which database system you use.

The transactional paradigm has its benefits and its drawbacks. Many users and application developers depend on the ease with which they can code around problems where an abort appears to be, or is necessary. However, even if you are new to the atomic operations paradigm, or more familiar with transactions, do consider the speed benefit that non-transactional tables can offer on the order of three to five times the speed of the fastest and most optimally tuned transactional tables.

In situations where integrity is of highest importance, MySQL Server offers transaction-level reliability and integrity even for non-transactional tables. If you lock tables with `LOCK TABLES`, all updates will stall until any integrity checks are made. If you obtain a `READ LOCAL` lock (as opposed to a write lock) for a table that allows concurrent inserts at the end of the table, reads are allowed, as are inserts by other clients. The new inserted records will not be seen by the client that has the read lock until it releases the lock. With `INSERT DELAYED`, you can queue inserts into a local queue, until the locks are released, without having the client wait for the insert to complete.

“Atomic,” in the sense that we mean it, is nothing magical. It only means that you can be sure that while each specific update is running, no other user can interfere with it, and there will never be an automatic rollback (which can happen with transactional tables if you are not very careful). MySQL Server also guarantees that there will not be any dirty reads.

Following are some techniques for working with non-transactional tables:

- Loops that need transactions normally can be coded with the help of `LOCK TABLES`, and you don’t need cursors to update records on the fly.
- To avoid using `ROLLBACK`, you can use the following strategy:
    1. Use `LOCK TABLES` to lock all the tables you want to access.
    2. Test the conditions that must be true before performing the update.
    3. Update if everything is okay.
    4. Use `UNLOCK TABLES` to release your locks.

  This is usually a much faster method than using transactions with possible rollbacks, although not always. The only situation this solution doesn’t handle is when someone kills the threads in the middle of an update. In this case, all locks will be released but some of the updates may not have been executed.

- You can also use functions to update records in a single operation. You can get a very efficient application by using the following techniques:
  - Modify columns relative to their current value.
  - Update only those columns that actually have changed.

  For example, when we are doing updates to some customer information, we update only the customer data that has changed and test only that none of the changed data, or data that depends on the changed data, has changed compared to the original row. The test for changed data is done with the `WHERE` clause in the `UPDATE` statement. If the record wasn't updated, we give the client a message: "Some of the data you have changed has been changed by another user." Then we show the old row versus the new row in a window so that the user can decide which version of the customer record to use.

  This gives us something that is similar to column locking but is actually even better because we only update some of the columns, using values that are relative to their current values. This means that typical `UPDATE` statements look something like these:

```
UPDATE tablename SET pay_back=pay_back+125;

UPDATE customer
  SET
    customer_date='current_date',
    address='new address',
    phone='new phone',
    money_owed_to_us=money_owed_to_us-125
  WHERE
    customer_id=id AND address='old address' AND phone='old phone';
```

  This is very efficient and works even if another client has changed the values in the `pay_back` or `money_owed_to_us` columns.

- In many cases, users have wanted `LOCK TABLES` and/or `ROLLBACK` for the purpose of managing unique identifiers. This can be handled much more efficiently without locking or rolling back by using an `AUTO_INCREMENT` column and either the `LAST_INSERT_ID()` SQL function or the `mysql_insert_id()` C API function.

  You can generally code around the need for row-level locking. Some situations really do need it, and `InnoDB` tables support row-level locking. With `MyISAM` tables, you can use a flag column in the table and do something like the following:

```
UPDATE tbl_name SET row_flag=1 WHERE id=ID;
```

  MySQL returns 1 for the number of affected rows if the row was found and `row_flag` wasn't already 1 in the original row.

  You can think of it as though MySQL Server changed the preceding query to:

```
UPDATE tbl_name SET row_flag=1 WHERE id=ID AND row_flag <> 1;
```

### 1.8.5.4 Stored Procedures and Triggers

Stored procedures are implemented in MySQL Version 5.0.

Triggers are scheduled for implementation in MySQL Version 5.1. A "trigger" is effectively a type of stored procedure, one that is invoked when a particular event occurs. For example, you could set up a stored procedure that is triggered each time a record is deleted from a transactional table, and that stored procedure automatically deletes the corresponding customer from a customer table when all their transactions are deleted.

### 1.8.5.5 Foreign Keys

In MySQL Server 3.23.44 and up, the `InnoDB` storage engine supports checking of foreign key constraints, including `CASCADE`, `ON DELETE`, and `ON UPDATE`.

For storage engines other than `InnoDB`, MySQL Server parses the `FOREIGN KEY` syntax in `CREATE TABLE` statements, but does not use or store it. In the future, the implementation will be extended to store this information in the table specification file so that it may be retrieved by `mysqldump` and ODBC. At a later stage, foreign key constraints will be implemented for `MyISAM` tables as well.

Foreign key enforcement offers several benefits to database developers:

- Assuming proper design of the relationships, foreign key constraints make it more difficult for a programmer to introduce an inconsistency into the database.
- Centralized checking of constraints by the database server makes it unnecessary to perform these checks on the application side. This eliminates the possibility that different applications may not all check the constraints in the same way.
- Using cascading updates and deletes can simplify the application code.
- Properly designed foreign key rules aid in documenting relationships between tables.

Do keep in mind that these benefits come at the cost of additional overhead for the database server to perform the necessary checks. Additional checking by the server affects performance, which for some applications may be sufficiently undesirable as to be avoided if possible. (Some major commercial applications have coded the foreign-key logic at the application level for this reason.)

MySQL gives database developers the choice of which approach to use. If you don't need foreign keys and want to avoid the overhead associated with enforcing referential integrity, you can choose another table type instead, such as `MyISAM`. (For example, the `MyISAM` storage engine offers very fast performance for applications that perform only `INSERT` and `SELECT` operations, because the inserts can be performed concurrently with retrievals.)

If you choose not to take advantage of referential integrity checks, keep the following considerations in mind:

- In the absence of server-side foreign key relationship checking, the application itself must handle relationship issues. For example, it must take care to insert rows into tables in the proper order, and to avoid creating orphaned child records. It must also be able to recover from errors that occur in the middle of multiple-record insert operations.
- If `ON DELETE` is the only referential integrity capability an application needs, note that as of MySQL Server 4.0, you can use multiple-table `DELETE` statements to delete rows from many tables with a single statement.
- A workaround for the lack of `ON DELETE` is to add the appropriate `DELETE` statement to your application when you delete records from a table that has a foreign key. In practice, this is often as quick as using foreign keys, and is more portable.

Be aware that the use of foreign keys can in some instances lead to problems:

- Foreign key support addresses many referential integrity issues, but it is still necessary to design key relationships carefully to avoid circular rules or incorrect combinations of cascading deletes.
- It is not uncommon for a DBA to create a topology of relationships that makes it difficult to restore individual tables from a backup. (MySQL alleviates this difficulty by allowing you to temporarily disable foreign key checks when reloading a table that depends on other tables. As of MySQL 4.1.1, `mysqldump` generates dump files that take advantage of this capability automatically when reloaded.)

Note that foreign keys in SQL are used to check and enforce referential integrity, not to join tables. If you want to get results from multiple tables from a `SELECT` statement, you do this by performing a join between them:

```
SELECT * FROM t1, t2 WHERE t1.id = t2.id;
```

The `FOREIGN KEY` syntax without `ON DELETE ...` is often used by ODBC applications to produce automatic `WHERE` clauses.

### 1.8.5.6 Views

Views currently are being implemented, and will appear in the 5.0 Version of MySQL Server. Unnamed views (*derived tables*, a subquery in the `FROM` clause of a `SELECT`) are already implemented in Version 4.1.

Historically, MySQL Server has been most used in applications and on Web systems where the application writer has full control over database usage. Usage has shifted over time, and so we find that an increasing number of users now regard views as an important feature.

Views are useful for allowing users to access a set of relations (tables) as if it were a single table, and limiting their access to just that. Views can also be used to restrict access to rows (a subset of a particular table). To restrict access to columns, views are not required because MySQL Server has a sophisticated privilege system.

Many DBMS don't allow updates to a view. Instead, you have to perform the updates on the individual tables. In designing an implementation of views, our goal, as much as is possible within the confines of SQL, is full compliance with "Codd's Rule #6" for relational database systems: All views that are theoretically updatable, should in practice also be updatable.

#### 1.8.5.7 '`--`' as the Start of a Comment

Some other SQL databases use '`--`' to start comments. MySQL Server uses '`#`' as the start comment character. You can also use the C comment style `/* this is a comment */` with MySQL Server.

MySQL Server 3.23.3 and above support the '`--`' comment style, provided the comment is followed by a space (or by a control character such as a newline). The requirement for a space is to prevent problems with automatically generated SQL queries that have used something like the following code, where we automatically insert the value of the payment for `!payment!`:

```
UPDATE account SET credit=credit-!payment!
```

Think about what happens if the value of `payment` is a negative value such as `-1`:

```
UPDATE account SET credit=credit--1
```

`credit--1` is a legal expression in SQL, but if `--` is interpreted as the start of a comment, part of the expression is discarded. The result is a statement that has a completely different meaning than intended:

```
UPDATE account SET credit=credit
```

The statement produces no change in value at all! This illustrates that allowing comments to start with '`--`' can have serious consequences.

Using our implementation of this method of commenting in MySQL Server 3.23.3 and up, `credit--1` is actually safe.

Another safe feature is that the `mysql` command-line client removes all lines that start with '`--`'.

The following information is relevant only if you are running a MySQL Version earlier than 3.23.3:

If you have an SQL program in a text file that contains '`--`' comments, you should use the `replace` utility as follows to convert the comments to use '`#`' characters:

```
shell> replace " --" " #" < text-file-with-funny-comments.sql \
        | mysql db_name
```

instead of the usual:

```
shell> mysql db_name < text-file-with-funny-comments.sql
```

You can also edit the command file "in place" to change the '--' comments to '#' comments:

```
shell> replace " --" " #" -- text-file-with-funny-comments.sql
```

Change them back with this command:

```
shell> replace " #" " --" -- text-file-with-funny-comments.sql
```

## 1.8.6 How MySQL Deals with Constraints

MySQL allows you to work with both transactional tables that allow rollback and non-transactional tables that do not, so constraint handling is a bit different in MySQL than in other databases.

We have to handle the case when you have updated a lot of rows in a non-transactional table that cannot roll back when an error occurs.

The basic philosophy is to try to give an error for anything that we can detect at compile time but try to recover from any errors we get at runtime. We do this in most cases, but not yet for all. See Section 1.6.4, "New Features Planned for the Near Future."

The options MySQL has when an error occurs are to stop the statement in the middle or to recover as well as possible from the problem and continue.

The following sections describe what happens for the different types of constraints.

### 1.8.6.1 Constraint `PRIMARY KEY` / `UNIQUE`

Normally, you will get an error when you try to `INSERT` or `UPDATE` a row that causes a primary key, unique key, or foreign key violation. If you are using a transactional storage engine such as `InnoDB`, MySQL will automatically roll back the transaction. If you are using a non-transactional storage engine, MySQL will stop at the incorrect row and leave any remaining rows unprocessed.

To make life easier, MySQL supports an `IGNORE` keyword for most commands that can cause a key violation (such as `INSERT IGNORE` and `UPDATE IGNORE`). In this case, MySQL will ignore any key violation and continue with processing the next row. You can get information about what MySQL did with the `mysql_info()` C API function. In MySQL 4.1 and up, you also can use the `SHOW WARNINGS` statement.

Note that, for the moment, only `InnoDB` tables support foreign keys. Foreign key support in `MyISAM` tables is scheduled for implementation in MySQL 5.1.

### 1.8.6.2 Constraint `NOT NULL` and `DEFAULT` Values

To be able to support easy handling of non-transactional tables, all columns in MySQL have default values.

If you insert an "incorrect" value in a column, such as a `NULL` in a `NOT NULL` column or a too-large numerical value in a numerical column, MySQL sets the column to the "best possible value" instead of producing an error:

- If you try to store a value outside the range in a numerical column, MySQL Server instead stores zero, the smallest possible value, or the largest possible value in the column.
- For strings, MySQL Server stores either the empty string or the longest possible string that can be in the column.
- If you try to store a string that doesn't start with a number into a numerical column, MySQL Server stores 0.
- If you try to store `NULL` into a column that doesn't take `NULL` values, MySQL Server stores 0 or `''` (the empty string) instead. This last behavior can, for single-row inserts, be changed when MySQL is built by using the `-DONT_USE_DEFAULT_FIELDS` compile option. This causes `INSERT` statements to generate an error unless you explicitly specify values for all columns that require a non-`NULL` value.
- MySQL allows you to store some incorrect date values into `DATE` and `DATETIME` columns (like `'2000-02-31'` or `'2000-02-00'`). The idea is that it's not the job of the SQL server to validate dates. If MySQL can store a date value and retrieve exactly the same value, MySQL stores it as given. If the date is totally wrong (outside the server's ability to store it), the special date value `'0000-00-00'` is stored in the column instead.

The reason for the preceding rules is that we can't check these conditions until the query has begun executing. We can't just roll back if we encounter a problem after updating a few rows, because the table type may not support rollback. The option of terminating the statement is not that good; in this case, the update would be "half done," which is probably the worst possible scenario. In this case, it's better to "do the best you can" and then continue as if nothing happened.

This means that you should generally not use MySQL to check column content. Instead, the application should ensure that it passes only legal values to MySQL.

In MySQL 5.0, we plan to improve this by providing warnings when automatic column conversions occur, plus an option to let you roll back statements that attempt to perform a disallowed column value assignment, as long as the statement uses only transactional tables.

### 1.8.6.3 Constraint `ENUM` and `SET`

In MySQL 4.x, `ENUM` is not a real constraint, but is a more efficient way to define columns that can contain only a given set of values. This is for same reasons that `NOT NULL` is not honored.

If you insert an incorrect value into an `ENUM` column, it is set to the reserved enumeration value of `0`, which is displayed as an empty string in string context.

If you insert an incorrect value into a `SET` column, the incorrect value is ignored. For example, if the column can contain the values `'a'`, `'b'`, and `'c'`, an attempt to assign `'a,x,b,y'` results in a value of `'a,b'`.

## 1.8.7 Known Errors and Design Deficiencies in MySQL

### 1.8.7.1 Errors in 3.23 Fixed in a Later MySQL Version

The following known errors or bugs are not fixed in MySQL 3.23 because fixing them would involve changing a lot of code that could introduce other even worse bugs. The bugs are also classified as "not fatal" or "bearable."

- You can get a deadlock (hung thread) if you use `LOCK TABLE` to lock multiple tables and then in the same connection use `DROP TABLE` to drop one of them while another thread is trying to lock it. (To break the deadlock, you can use `KILL` to terminate any of the threads involved.) This issue is resolved as of MySQL 4.0.12.
- `SELECT MAX(`*`key_column`*`) FROM t1,t2,t3...` where one of the tables is empty doesn't return `NULL` but instead returns the maximum value for the column. This issue is resolved as of MySQL 4.0.11.
- `DELETE FROM` *`heap_table`* without a `WHERE` clause doesn't work on a locked `HEAP` table.

### 1.8.7.2 Errors in 4.0 Fixed in a Later MySQL Version

The following known errors or bugs are not fixed in MySQL 4.0 because fixing them would involve changing a lot of code that could introduce other even worse bugs. The bugs are also classified as "not fatal" or "bearable."

- In a `UNION`, the first `SELECT` determines the type, `max_length`, and `NULL` properties for the resulting columns. This issue is resolved as of MySQL 4.1.1; the property values are based on the rows from all `UNION` parts.
- In `DELETE` with many tables, you can't refer to tables to be deleted through an alias. This is fixed as of MySQL 4.1.

### 1.8.7.3 Open Bugs and Design Deficiencies in MySQL

The following problems are known and fixing them is a high priority:

- Dropping a `FOREIGN KEY` constraint doesn't work in replication because the constraint may have another name on the slave.
- `REPLACE` (and `LOAD DATA` with the `REPLACE` option) does not trigger `ON DELETE CASCADE`.
- You cannot mix `UNION ALL` and `UNION DISTINCT` in the same query. If you use `ALL` for one `UNION`, it is used for all of them.
- If one user has a long-running transaction and another user drops a table that is updated in the transaction, there is small chance that the binary log may contain the `DROP TABLE` command before the table is used in the transaction itself. We plan to fix this in 5.0 by having the `DROP TABLE` wait until the table is not used in any transaction.
- When inserting a big integer value (between $2^{63}$ and $2^{64}–1$) into a decimal/string column, it is inserted as a negative value because the number is evaluated in a signed integer context. We plan to fix this in MySQL 4.1.
- `FLUSH TABLES WITH READ LOCK` does not block `CREATE TABLE` or `COMMIT`, which may cause a problem with the binary log position when doing a full backup of tables and the binary log.
- `ANALYZE TABLE` on a `BDB` table may in some cases make the table unusable until you restart `mysqld`. If this happens, you will see errors of the following form in the MySQL error file:

  ```
  001207 22:07:56  bdb:  log_flush: LSN past current end-of-log
  ```

- MySQL accepts parentheses in the `FROM` clause of a `SELECT` statement, but silently ignores them. The reason for not giving an error is that many clients that automatically generate queries add parentheses in the `FROM` clause even where they are not needed.
- Concatenating many `RIGHT JOINS` or combining `LEFT` and `RIGHT` join in the same query may not give a correct answer because MySQL only generates `NULL` rows for the table preceding a `LEFT` or before a `RIGHT` join. This will be fixed in 5.0 at the same time we add support for parentheses in the `FROM` clause.
- Don't execute `ALTER TABLE` on a `BDB` table on which you are running multiple-statement transactions until all those transactions complete. (The transaction will probably be ignored.)
- `ANALYZE TABLE`, `OPTIMIZE TABLE`, and `REPAIR TABLE` may cause problems on tables for which you are using `INSERT DELAYED`.
- Doing a `LOCK TABLE ...` and `FLUSH TABLES ...` doesn't guarantee that there isn't a half-finished transaction in progress on the table.
- `BDB` tables are a bit slow to open. If you have many `BDB` tables in a database, it will take a long time to use the `mysql` client on the database if you are not using the `-A` option or if you are using `rehash`. This is especially noticeable when you have a large table cache.

- Replication uses query-level logging: The master writes the executed queries to the binary log. This is a very fast, compact, and efficient logging method that works perfectly in most cases. Although we have never heard of it actually occurring, it is theoretically possible for the data on the master and slave to become different if a query is designed in such a way that the data modification is non-deterministic; that is, left to the will of the query optimizer. (That generally is not a good practice anyway, even outside of replication!) For example:
    - `CREATE ... SELECT` or `INSERT ... SELECT` statements that insert zero or `NULL` values into an `AUTO_INCREMENT` column.
    - `DELETE` if you are deleting rows from a table that has foreign keys with `ON DELETE CASCADE` properties.
    - `REPLACE ... SELECT`, `INSERT IGNORE ... SELECT` if you have duplicate key values in the inserted data.

  **If and only if all these queries have no `ORDER BY` clause guaranteeing a deterministic order.**

  For example, for `INSERT ... SELECT` with no `ORDER BY`, the `SELECT` may return rows in a different order (which will result in a row having different ranks, hence getting a different number in the `AUTO_INCREMENT` column), depending on the choices made by the optimizers on the master and slave. A query will be optimized differently on the master and slave only if:
    - The files used by the two queries are not exactly the same; for example, `OPTIMIZE TABLE` was run on the master tables and not on the slave tables. (To fix this, `OPTIMIZE TABLE`, `ANALYZE TABLE`, and `REPAIR TABLE` are written to the binary log as of MySQL 4.1.1.)
    - The table is stored using a different storage engine on the master than on the slave. (It is possible to use different storage engines on the master and slave. For example, you can use `InnoDB` on the master, but `MyISAM` on the slave if the slave has less available disk space.)
    - MySQL buffer sizes (`key_buffer_size`, and so on) are different on the master and slave.
    - The master and slave run different MySQL versions, and the optimizer code differs between these versions.

  This problem may also affect database restoration using `mysqlbinlog|mysql`.

  The easiest way to avoid this problem in all cases is to add an `ORDER BY` clause to such non-deterministic queries to ensure that the rows are always stored or modified in the same order. In future MySQL versions, we will automatically add an `ORDER BY` clause when needed.

The following problems are known and will be fixed in due time:

- Log filenames are based on the server hostname (if you don't specify a filename with the startup option). For now you have to use options like `--log-bin=`*`old_host_name`*`-bin` if you change your hostname to something else. Another option is to just rename the old files to reflect your hostname change.
- `mysqlbinlog` will not delete temporary files left after a `LOAD DATA INFILE` command.
- `RENAME` doesn't work with `TEMPORARY` tables or tables used in a `MERGE` table.
- When using the `RPAD()` function in a query that has to be resolved by using a temporary table, all resulting strings will have rightmost spaces removed. This is an example of such a query:

  ```
  SELECT RPAD(t1.column1, 50, ' ') AS f2, RPAD(t2.column2, 50, ' ') AS f1
  FROM table1 as t1 LEFT JOIN table2 AS t2 ON t1.record=t2.joinID
  ORDER BY t2.record;
  ```

  The final result of this bug is that you will not be able to get spaces on the right side of the resulting values. The problem also occurs for any other string function that adds spaces to the right.

  The reason for this is due to the fact that `HEAP` tables, which are used first for temporary tables, are not capable of handling `VARCHAR` columns.

  This behavior exists in all versions of MySQL. It will be fixed in one of the 4.1 series releases.
- Due to the way table definition files are stored, you cannot use character 255 (`CHAR(255)`) in table names, column names, or enumerations. This is scheduled to be fixed in Version 5.1 when we have new table definition format files.
- When using `SET CHARACTER SET`, you can't use translated characters in database, table, and column names.
- You can't use '_' or '%' with `ESCAPE` in `LIKE ... ESCAPE`.
- If you have a `DECIMAL` column in which the same number is stored in different formats (for example, `+01.00`, `1.00`, `01.00`), `GROUP BY` may regard each value as a different value.
- You cannot build the server in another directory when using MIT-pthreads. Because this requires changes to MIT-pthreads, we are not likely to fix this.
- `BLOB` values can't "reliably" be used in `GROUP BY` or `ORDER BY` or `DISTINCT`. Only the first `max_sort_length` bytes are used when comparing `BLOB` values in these cases. The default value of `max_sort_length` value is 1024. It can be changed at server startup time. A workaround for most cases is to use a substring. For example: `SELECT DISTINCT LEFT(`*`blob_col`*`,2048) FROM `*`tbl_name`*.
- Numeric calculations are done with `BIGINT` or `DOUBLE` (both are normally 64 bits long). Which precision you get depends on the function. The general rule is that bit functions are done with `BIGINT` precision, `IF` and `ELT()` with `BIGINT` or `DOUBLE` precision, and the

rest with DOUBLE precision. You should try to avoid using unsigned long long values if they resolve to be bigger than 63 bits (9223372036854775807) for anything other than bit fields. MySQL Server 4.0 has better BIGINT handling than 3.23.

- All string columns, except BLOB and TEXT columns, automatically have all trailing spaces removed when retrieved. For CHAR types, this is okay. The bug is that in MySQL Server, VARCHAR columns are treated the same way.
- You can have only up to 255 ENUM and SET columns in one table.
- In MIN(), MAX(), and other aggregate functions, MySQL currently compares ENUM and SET columns by their string value rather than by the string's relative position in the set.
- mysqld_safe redirects all messages from mysqld to the mysqld log. One problem with this is that if you execute mysqladmin refresh to close and reopen the log, stdout and stderr are still redirected to the old log. If you use --log extensively, you should edit mysqld_safe to log to *host_name*.err instead of *host_name*.log so that you can easily reclaim the space for the old log by deleting the old one and executing mysqladmin refresh.
- In the UPDATE statement, columns are updated from left to right. If you refer to an updated column, you get the updated value instead of the original value. For example, the following statement increments KEY by 2, not 1:

```
mysql> UPDATE tbl_name SET KEY=KEY+1,KEY=KEY+1;
```

- You can refer to multiple temporary tables in the same query, but you cannot refer to any given temporary table more than once. For example, the following doesn't work:

```
mysql> SELECT * FROM temp_table, temp_table AS t2;
ERROR 1137: Can't reopen table: 'temp_table'
```

- The optimizer may handle DISTINCT differently when you are using "hidden" columns in a join than when you are not. In a join, hidden columns are counted as part of the result (even if they are not shown), whereas in normal queries, hidden columns don't participate in the DISTINCT comparison. We will probably change this in the future to never compare the hidden columns when executing DISTINCT.

  An example of this is:

```
SELECT DISTINCT mp3id FROM band_downloads
       WHERE userid = 9 ORDER BY id DESC;
```

  and

```
SELECT DISTINCT band_downloads.mp3id
       FROM band_downloads,band_mp3
       WHERE band_downloads.userid = 9
       AND band_mp3.id = band_downloads.mp3id
       ORDER BY band_downloads.id DESC;
```

In the second case, you might in MySQL Server 3.23.x get two identical rows in the result set (because the values in the hidden `id` column may differ).

Note that this happens only for queries where you don't have the `ORDER BY` columns in the result.

- Because MySQL Server allows you to work with table types that don't support transactions, and thus can't roll back data, some things behave a little differently in MySQL Server than in other SQL servers. This is just to ensure that MySQL Server never needs to do a rollback for an SQL statement. This may be a little awkward at times because column values must be checked in the application, but this will actually give you a nice speed increase because it allows MySQL Server to do some optimizations that otherwise would be very hard to do.

  If you set a column to an incorrect value, MySQL Server will, instead of doing a rollback, store the "best possible value" in the column. For information about how this occurs, see Section 1.8.6, "How MySQL Deals with Constraints."
- If you execute a `PROCEDURE` on a query that returns an empty set, in some cases the `PROCEDURE` will not transform the columns.
- Creation of a table of type `MERGE` doesn't check whether the underlying tables are of compatible types.
- If you use `ALTER TABLE` first to add a `UNIQUE` index to a table used in a `MERGE` table and then to add a normal index on the `MERGE` table, the key order will be different for the tables if there was an old key that was not unique in the table. This is because `ALTER TABLE` puts `UNIQUE` indexes before normal indexes to be able to detect duplicate keys as early as possible.

The following are known bugs in earlier versions of MySQL:

- In the following case you can get a core dump:
  - Delayed insert handler has pending inserts to a table.
  - `LOCK TABLE` with `WRITE`.
  - `FLUSH TABLES`.
- Before MySQL Server 3.23.2, an `UPDATE` that updated a key with a `WHERE` on the same key may have failed because the key was used to search for records and the same row may have been found multiple times:

  ```
  UPDATE tbl_name SET KEY=KEY+1 WHERE KEY > 100;
  ```

  A workaround is to use:

  ```
  UPDATE tbl_name SET KEY=KEY+1 WHERE KEY+0 > 100;
  ```

  This will work because MySQL Server will not use an index on expressions in the `WHERE` clause.
- Before MySQL Server 3.23, all numeric types were treated as fixed-point fields. That means you had to specify how many decimals a floating-point field should have. All results were returned with the correct number of decimals.

# 2

# Language Structure

This chapter discusses the rules for writing the following elements of SQL statements when using MySQL:

- Literal values such as strings and numbers
- Identifiers such as table and column names
- User and system variables
- Comments
- Reserved words

## 2.1 Literal Values

This section describes how to write literal values in MySQL. These include strings, numbers, hexadecimal values, boolean values, and `NULL`. The section also covers the various nuances and "gotchas" that you may run into when dealing with these basic types in MySQL.

### 2.1.1 Strings

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

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

If the server SQL mode has `ANSI_QUOTES` enabled, string literals can be quoted only with single quotes. A string quoted with double quotes will be interpreted as an identifier.

As of MySQL 4.1.1, string literals may have an optional character set introducer and `COLLATE` clause:

```
[_charset_name]'string' [COLLATE collation_name]
```

Examples:

```
SELECT _latin1'string';
SELECT _latin1'string' COLLATE latin1_danish_ci;
```

For more information about these forms of string syntax, see Section 3.3.7, "Character String Literal Character Set and Collation."

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. |
| `\'` | A single quote (`'''`) character. |
| `\"` | A double quote (`'"'`) character. |
| `\b` | A backspace character. |
| `\n` | A newline (linefeed) character. |
| `\r` | A carriage return character. |
| `\t` | A tab character. |
| `\z` | ASCII 26 (Control-Z). This character can be encoded as `'\z'` to allow you to work around the problem that ASCII 26 stands for END-OF-FILE on Windows. (ASCII 26 will cause problems if you try to use `mysql` *`db_name`* `<` *`file_name`*.) |
| `\\` | A backslash (`'\'`) character. |
| `\%` | A `'%'` character. See note following table. |
| `\_` | A `'_'` character. See note following table. |

These sequences are case sensitive. For example, '`\b`' is interpreted as a backslash, but '`\B`' is interpreted as '`B`'.

The '`\%`' and '`\_`' sequences are used to search for literal instances of '`%`' and '`_`' in pattern-matching contexts where they would otherwise be interpreted as wildcard characters. See Section 5.3.1, "String Comparison Functions." Note that if you use '`\%`' or '`\_`' in other contexts, they return the strings '`\%`' and '`\_`' and not '`%`' and '`_`'.

In all other escape sequences, backslash is ignored. That is, the escaped character is interpreted as if it was not escaped.

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 following SELECT statements 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 string column (such as a BLOB), the following characters must be represented by escape sequences:

| | |
|---|---|
| NUL | NUL byte (ASCII 0). Represent this character by '\0' (a backslash followed by an ASCII '0' character). |
| \ | Backslash (ASCII 92). Represent this character by '\\'. |
| ' | Single quote (ASCII 39). Represent this character by '\''. |
| " | Double quote (ASCII 34). Represent this character by '\"'. |

When writing application programs, any string that might contain any of these special characters must be properly escaped before the string is used as a data value in an SQL statement that is sent to the MySQL server. You can do this in two ways:

- Process the string with a function that escapes the special characters. For example, in a C program, you can use the `mysql_real_escape_string()` C API function to escape characters. The Perl DBI interface provides a `quote` method to convert special characters to the proper escape sequences.
- As an alternative to explicitly escaping special characters, many MySQL APIs provide a placeholder capability that allows you to insert special markers into a query string, and then bind data values to them when you issue the query. In this case, the API takes care of escaping special characters in the values for you.

## 2.1.2 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.

## 2.1.3 Hexadecimal Values

MySQL supports hexadecimal values. In numeric contexts, these act like integers (64-bit precision). In string contexts, these act like binary strings, where each pair of hex digits is converted to a character:

```
mysql> SELECT x'4D7953514C';
        -> 'MySQL'
mysql> SELECT 0xa+0;
        -> 10
mysql> SELECT 0x5061756c;
        -> 'Paul'
```

In MySQL 4.1 (and in MySQL 4.0 when using the `--new` option), the default type of a hexadecimal value is a string. If you want to ensure that the value is treated as a number, you can use `CAST(... AS UNSIGNED)`:

```
mysql> SELECT 0x41, CAST(0x41 AS UNSIGNED);
        -> 'A', 65
```

The `0x` syntax is based on ODBC. Hexadecimal strings are often used by ODBC to supply values for `BLOB` columns. The `x'`*`hexstring`*`'` syntax is new in 4.0 and is based on standard SQL.

Beginning with MySQL 4.0.1, you can convert a string or a number to a string in hexadecimal format with the `HEX()` function:

```
mysql> SELECT HEX('cat');
        -> '636174'
mysql> SELECT 0x636174;
        -> 'cat'
```

### 2.1.4 Boolean Values

Beginning with MySQL 4.1, the constant `TRUE` evaluates to `1` and the constant `FALSE` evaluates to `0`. The constant names can be written in any lettercase.

```
mysql> SELECT TRUE, true, FALSE, false;
        -> 1, 1, 0, 0
```

### 2.1.5 `NULL` Values

The `NULL` value means "no data." `NULL` can be written in any lettercase.

Be aware that the `NULL` value is different from values such as `0` for numeric types or the empty string for string types. See section A.1.3, "Problems with `NULL` Values."

For text file import or export operations performed with `LOAD DATA INFILE` or `SELECT ... INTO OUTFILE`, `NULL` is represented by the `\N` sequence. See Section 6.1.5, "`LOAD DATA INFILE` Syntax."

## 2.2 Database, Table, Index, Column, and Alias Names

Database, table, index, column, and alias names are identifiers. This section describes the allowable syntax for identifiers in MySQL.

The following table describes the maximum length and allowable characters for each type of identifier.

| Identifier | Maximum Length (bytes) | Allowed Characters |
|---|---|---|
| Database | 64 | Any character that is allowed in a directory name' except '/', '\', or '.' |
| Table | 64 | Any character that is allowed in a filename, except '/', '\', or '.' |
| Column | 64 | All characters |
| Index | 64 | All characters |
| Alias | 255 | All characters |

In addition to the restrictions noted in the table, no identifier can contain ASCII 0 or a byte with a value of 255. Database, table, and column names should not end with space characters. Before MySQL 4.1, identifier quote characters should not be used in identifiers.

Beginning with MySQL 4.1, identifiers are stored using Unicode (UTF8). This applies to identifiers in table definitions stored in `.frm` files and to identifiers stored in the grant tables in the `mysql` database. Although Unicode identifiers can include multi-byte characters, note that the maximum lengths shown in the table are byte counts. If an identifier does contain multi-byte characters, the number of *characters* allowed in the identifier is less than the value shown in the table.

An identifier may be quoted or unquoted. If an identifier is a reserved word or contains special characters, you *must* quote it whenever you refer to it. For a list of reserved words, see Section 2.6, "Treatment of Reserved Words in MySQL." Special characters are those outside the set of alphanumeric characters from the current character set, '_', and '$'.

The identifier quote character is the backtick ('`'):

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

If the server SQL mode includes the `ANSI_QUOTES` mode option, it is also allowable to quote identifiers with double quotes:

```
mysql> CREATE TABLE "test" (col INT);
ERROR 1064: You have an error in your SQL syntax. (...)
mysql> SET sql_mode='ANSI_QUOTES';
mysql> CREATE TABLE "test" (col INT);
Query OK, 0 rows affected (0.00 sec)
```

See section 1.8.2, "Selecting SQL Modes."

As of MySQL 4.1, identifier quote characters can be included within an identifier by quoting the identifier. If the character to be included within the identifier is the same as that used to quote the identifier itself, double the character. The following statement creates a table named ``a`b`` that contains a column named `c"d`:

```
mysql> CREATE TABLE `a``b` (`c"d` INT);
```

Identifier quoting was introduced in MySQL 3.23.6 to allow use of identifiers that are reserved words or that contain special characters. Before 3.23.6, you cannot use identifiers that require quotes, so the rules for legal identifiers are more restrictive:

- A name may consist of alphanumeric characters from the current character set, '_', and '$'. The default character set is ISO-8859-1 (Latin1). This may be changed with the `--default-character-set` option to `mysqld`.
- A name may start with any character that is legal in a name. In particular, a name may start with a digit; this differs from many other database systems! However, an unquoted name cannot consist *only* of digits.
- You cannot use the '.' character in names because it is used to extend the format by which you can refer to columns (see Section 2.2.1, "Identifier Qualifiers").

It is recommended that you do not use names like `1e`, because an expression like `1e+1` is ambiguous. It might be interpreted as the expression `1e + 1` or as the number `1e+1`, depending on context.

### 2.2.1 Identifier Qualifiers

MySQL allows names that consist of a single identifier or multiple identifiers. The components of a multiple-part name should be separated by period ('`.`') characters. The initial parts of a multiple-part name act as qualifiers that affect the context within which the final identifier is interpreted.

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

| Column Reference | Meaning |
|---|---|
| *`col_name`* | The column *`col_name`* from whichever table used in the query contains a column of that name. |
| *`tbl_name.col_name`* | The column *`col_name`* from table *`tbl_name`* of the default database. |
| *`db_name.tbl_name.col_name`* | The column *`col_name`* from table *`tbl_name`* of the database *`db_name`*. This syntax is unavailable before MySQL 3.22. |

If any components of a multiple-part name require quoting, quote them individually rather than quoting the name as a whole. For example, `` `my-table`.`my-column` `` is legal, whereas `` `my-table.my-column` `` is not.

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. Suppose that 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. You must qualify it with a table name as `t1.c` or `t2.c` to indicate which table you mean. Similarly, to retrieve from a table `t` in database `db1` and from a table `t` in database `db2` in the same statement, 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.

### 2.2.2 Identifier Case Sensitivity

In MySQL, databases correspond to directories within the data directory. Tables within a database correspond to at least one file within the database directory (and possibly more, depending on the storage engine). 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 not case sensitive in Windows, and case sensitive in most varieties of Unix. One notable exception is Mac OS X, which is Unix-based but uses a default filesystem type (HFS+) that is not case sensitive. However, Mac OS X also supports UFS volumes, which are case sensitive just as on any Unix. See section 1.8.4, "MySQL Extensions to Standard SQL."

**Note:** Although database and table names are not case sensitive on some platforms, 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`:

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

Column names, index names, and column aliases are not case sensitive on any platform.

Table aliases are case sensitive before MySQL 4.1.1. 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;
```

If you have trouble remembering the allowable lettercase for database and table names, adopt a consistent convention, such as always creating databases and tables using lowercase names.

How table names are stored on disk and used in MySQL is defined by the `lower_case_table_names` system variable, which you can set when starting `mysqld`. `lower_case_table_names` can take one of the following values:

| Value | Meaning |
|---|---|
| 0 | Table and database names are stored on disk using the lettercase specified in the `CREATE TABLE` or `CREATE DATABASE` statement. Name comparisons are case sensitive. This is the default on Unix systems. Note that if you force this to 0 with `--lower-case-table-names=0` on a case-insensitive filesystem and access `MyISAM` table names using different lettercases, this may lead to index corruption. |
| 1 | Table names are stored in lowercase on disk and name comparisons are not case sensitive. MySQL converts all table names to lowercase on storage and lookup. This behavior also applies to database names as of MySQL 4.0.2, and to table aliases as of 4.1.1. This value is the default on Windows and Mac OS X systems. |
| 2 | Table and database names are stored on disk using the lettercase specified in the `CREATE TABLE` or `CREATE DATABASE` statement, but MySQL converts them to lowercase on lookup. Name comparisons are not case sensitive. **Note:** This works *only* on filesystems that are not case sensitive! `InnoDB` table names are stored in lowercase, as for `lower_case_table_names=1`. Setting `lower_case_table_names` to `2` can be done as of MySQL 4.0.18. |

If you are using MySQL on only one platform, you don't normally have to change the `lower_case_table_names` variable. However, you may encounter difficulties if you want to transfer tables between platforms that differ in filesystem case sensitivity. For example, on Unix, you can have two different tables named `my_table` and `MY_TABLE`, but on Windows those names are considered the same. To avoid data transfer problems stemming from database or table name lettercase, you have two options:

- Use `lower_case_table_names=1` on all systems. The main disadvantage with this is that when you use `SHOW TABLES` or `SHOW DATABASES`, you don't see the names in their original lettercase.

- Use `lower_case_table_names=0` on Unix and `lower_case_table_names=2` on Windows. This preserves the lettercase of database and table names. The disadvantage of this is that you must ensure that your queries always refer to your database and table names with the correct lettercase on Windows. If you transfer your queries to Unix, where lettercase is significant, they will not work if the lettercase is incorrect.

Note that before setting `lower_case_table_names` to 1 on Unix, you must first convert your old database and table names to lowercase before restarting `mysqld`.

## 2.3 User Variables

MySQL supports user variables as of version 3.23.6. You can store a value in a user variable and refer to it later, which allows you to pass values from one statement to another. User variables are connection-specific. That is, a variable defined by one client cannot be seen or used by other clients. All variables for a client connection are automatically freed when the client exits.

User variables are written as `@`*`var_name`*, where the variable name *`var_name`* may consist of alphanumeric characters from the current character set, '`.`', '`_`', and '`$`'. The default character set is ISO-8859-1 (Latin1). This may be changed with the `--default-character-set` option to `mysqld`. User variable names are not case sensitive beginning with MySQL 5.0. Before that, they are case sensitive.

One way to set a user variable is by issuing a `SET` statement:

```
SET @var_name = expr [, @var_name = expr] ...
```

For `SET`, either `=` or `:=` can be used as the assignment operator. The *`expr`* assigned to each variable can evaluate to an integer, real, string, or `NULL` value.

You can also assign a value to a user variable in statements other than `SET`. In this case, the assignment operator must be `:=` and not `=` because `=` is treated as a comparison operator in non-`SET` statements:

```
mysql> SET @t1=0, @t2=0, @t3=0;
mysql> SELECT @t1:=(@t2:=1)+@t3:=4,@t1,@t2,@t3;
+----------------------+------+------+------+
| @t1:=(@t2:=1)+@t3:=4 | @t1  | @t2  | @t3  |
+----------------------+------+------+------+
|                    5 |    5 |    1 |    4 |
+----------------------+------+------+------+
```

User variables may be used where expressions are allowed. This does not currently include contexts that explicitly require a number, such as in the `LIMIT` clause of a `SELECT` statement, or the `IGNORE` *`number`* `LINES` clause of a `LOAD DATA` statement.

If you refer to a variable that has not been initialized, its value is `NULL`.

**Note:** In a `SELECT` statement, each expression is evaluated only when sent to the client. This means that in a `HAVING`, `GROUP BY`, or `ORDER BY` clause, you cannot refer to an expression that involves variables that are set in the `SELECT` list. For example, the following statement will *not* work as expected:

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

The reference to `b` in the `HAVING` clause refers to an alias for an expression in the `SELECT` list that uses `@aa`. This does not work as expected: `@aa` will not contain the value of the current row, but the value of `id` from the previous selected row.

The general rule is to never assign *and* use the same variable in the same statement.

Another issue with setting a variable and using it in the same statement is that the default result type of a variable is based on the type of the variable at the start of the statement. The following example illustrates this:

```
mysql> SET @a='test';
mysql> SELECT @a,(@a:=20) FROM tbl_name;
```

For this `SELECT` statement, MySQL will report to the client that column one is a string and convert all accesses of `@a` to strings, even though `@a` is set to a number for the second row. After the `SELECT` statement executes, `@a` will be regarded as a number for the next statement.

To avoid problems with this behavior, either do not set and use the same variable within a single statement, or else set the variable to `0`, `0.0`, or `''` to define its type before you use it.

An unassigned variable has a value of `NULL` with a type of string.

## 2.4 System Variables

Starting from MySQL 4.0.3, we provide better access to a lot of system and connection variables. Many variables can be changed dynamically while the server is running. This allows you to modify server operation without having to stop and restart it.

The `mysqld` server maintains two kinds of variables. Global variables affect the overall operation of the server. Session variables affect its operation for individual client connections.

When the server starts, it initializes all global variables to their default values. These defaults may be changed by options specified in option files or on the command line. After the server starts, those global variables that are dynamic can be changed by connecting to the server and issuing a `SET GLOBAL` *`var_name`* statement. To change a global variable, you must have the `SUPER` privilege.

The server also maintains a set of session variables for each client that connects. The client's session variables are initialized at connect time using the current values of the corresponding global variables. For those session variables that are dynamic, the client can change them by issuing a `SET SESSION` *`var_name`* statement. Setting a session variable requires no special privilege, but a client can change only its own session variables, not those of any other client.

A change to a global variable is visible to any client that accesses that global variable. However, it affects the corresponding session variable that is initialized from the global variable only for clients that connect after the change. It does not affect the session variable for any client that is already connected (not even that of the client that issues the `SET GLOBAL` statement).

Global or session variables may be set or retrieved using several syntax forms. The following examples use `sort_buffer_size` as a sample variable name.

To set the value of a `GLOBAL` variable, use one of the following syntaxes:

```
mysql> SET GLOBAL sort_buffer_size=value;
mysql> SET @@global.sort_buffer_size=value;
```

To set the value of a `SESSION` variable, use one of the following syntaxes:

```
mysql> SET SESSION sort_buffer_size=value;
mysql> SET @@session.sort_buffer_size=value;
mysql> SET sort_buffer_size=value;
```

`LOCAL` is a synonym for `SESSION`.

If you don't specify `GLOBAL`, `SESSION`, or `LOCAL` when setting a variable, `SESSION` is the default. See Section 6.5.3.1, "`SET` Syntax."

To retrieve the value of a `GLOBAL` variable, use one of the following statements:

```
mysql> SELECT @@global.sort_buffer_size;
mysql> SHOW GLOBAL VARIABLES like 'sort_buffer_size';
```

To retrieve the value of a `SESSION` variable, use one of the following statements:

```
mysql> SELECT @@sort_buffer_size;
mysql> SELECT @@session.sort_buffer_size;
mysql> SHOW SESSION VARIABLES like 'sort_buffer_size';
```

Here, too, `LOCAL` is a synonym for `SESSION`.

When you retrieve a variable with `SELECT @@var_name` (that is, you do not specify `global.`, `session.`, or `local.`), MySQL returns the `SESSION` value if it exists and the `GLOBAL` value otherwise.

For `SHOW VARIABLES`, if you do not specify `GLOBAL`, `SESSION`, or `LOCAL`, MySQL returns the `SESSION` value.

The reason for requiring the `GLOBAL` keyword when setting `GLOBAL`-only variables but not when retrieving them is to prevent problems in the future. If we remove a `SESSION` variable with the same name as a `SESSION` variable, a client with the `SUPER` privilege might accidentally change the `GLOBAL` variable rather than just the `SESSION` variable for its own connection. If we add a `SESSION` variable with the same name as a `SESSION` variable, a client that intends to change the `GLOBAL` variable might find only its own `SESSION` variable changed.

Further information about system startup options and system variables can be found in the *MySQL Administrator's Guide*. A list of the variables that can be set at runtime is given there as well.

## 2.4.1 Structured System Variables

Structured system variables are supported beginning with MySQL 4.1.1. A structured variable differs from a regular system variable in two respects:

- Its value is a structure with components that specify server parameters considered to be closely related.
- There might be several instances of a given type of structured variable. Each one has a different name and refers to a different resource maintained by the server.

Currently, MySQL supports one structured variable type. It specifies parameters that govern the operation of key caches. A key cache structured variable has these components:

- `key_buffer_size`
- `key_cache_block_size`
- `key_cache_division_limit`
- `key_cache_age_threshold`

The purpose of this section is to describe the syntax for referring to structured variables. Key cache variables are used for syntax examples, but specific details about how key caches operate are found in the *MySQL Administrator's Guide*.

To refer to a component of a structured variable instance, you can use a compound name in *`instance_name.component_name`* format. Examples:

```
hot_cache.key_buffer_size
hot_cache.key_cache_block_size
cold_cache.key_cache_block_size
```

For each structured system variable, an instance with the name of `default` is always predefined. If you refer to a component of a structured variable without any instance name, the `default` instance is used. Thus, `default.key_buffer_size` and `key_buffer_size` both refer to the same system variable.

The naming rules for structured variable instances and components are as follows:

- For a given type of structured variable, each instance must have a name that is unique *within* variables of that type. However, instance names need not be unique *across* structured variable types. For example, each structured variable will have an instance named `default`, so `default` is not unique across variable types.
- The names of the components of each structured variable type must be unique across all system variable names. If this were not true (that is, if two different types of structured variables could share component member names), it would not be clear which default structured variable to use for references to member names that are not qualified by an instance name.

- If a structured variable instance name is not legal as an unquoted identifier, refer to it as a quoted identifier using backticks. For example, `hot-cache` is not legal, but `` `hot-cache` `` is.
- `global`, `session`, and `local` are not legal instance names. This avoids a conflict with notation such as `@@global.`*`var_name`* for referring to non-structured system variables.

At the moment, the first two rules have no possibility of being violated because the only structured variable type is the one for key caches. These rules will assume greater significance if some other type of structured variable is created in the future.

With one exception, it is allowable to refer to structured variable components using compound names in any context where simple variable names can occur. For example, you can assign a value to a structured variable using a command-line option:

```
shell> mysqld --hot_cache.key_buffer_size=64K
```

In an option file, do this:

```
[mysqld]
hot_cache.key_buffer_size=64K
```

If you start the server with such an option, it creates a key cache named `hot_cache` with a size of 64KB in addition to the default key cache that has a default size of 8MB.

Suppose that you start the server as follows:

```
shell> mysqld --key_buffer_size=256K \
         --extra_cache.key_buffer_size=128K \
         --extra_cache.key_cache_block_size=2048
```

In this case, the server sets the size of the default key cache to 256KB. (You could also have written `--default.key_buffer_size=256K`.) In addition, the server creates a second key cache named `extra_cache` that has a size of 128KB, with the size of block buffers for caching table index blocks set to 2048 bytes.

The following example starts the server with three different key caches having sizes in a 3:1:1 ratio:

```
shell> mysqld --key_buffer_size=6M \
         --hot_cache.key_buffer_size=2M \
         --cold_cache.key_buffer_size=2M
```

Structured variable values may be set and retrieved at runtime as well. For example, to set a key cache named `hot_cache` to a size of 10MB, use either of these statements:

```
mysql> SET GLOBAL hot_cache.key_buffer_size = 10*1024*1024;
mysql> SET @@global.hot_cache.key_buffer_size = 10*1024*1024;
```

To retrieve the cache size, do this:

```
mysql> SELECT @@global.hot_cache.key_buffer_size;
```

However, the following statement does not work. The variable is not interpreted as a compound name, but as a simple string for a `LIKE` pattern-matching operation:

```
mysql> SHOW GLOBAL VARIABLES LIKE 'hot_cache.key_buffer_size';
```

This is the exception to being able to use structured variable names anywhere a simple variable name may occur.

## 2.5 Comment Syntax

The MySQL server supports three comment styles:

- From a '`#`' character to the end of the line.
- From a '`-- `' sequence to the end of the line. This style is supported as of MySQL 3.23.3. Note that the '`-- `' (double-dash) comment style requires the second dash to be followed by at least one space (or by a control character such as a newline). This syntax differs slightly from standard SQL comment syntax, as discussed in section 1.8.5.7, " '`--`' as the Start of a Comment."
- From a '`/*`' sequence to the following '`*/`' sequence. The closing sequence need not be on the same line, so this syntax allows a comment to extend over multiple lines.

The following example demonstrates all three comment styles:

```
mysql> SELECT 1+1;     # This comment continues to the end of line
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;
```

The comment syntax just described applies to how the `mysqld` server parses SQL statements. The `mysql` client program also performs some parsing of statements before sending them to the server. (For example, it does this to determine statement boundaries within a multiple-statement input line.) However, there are some limitations on the way that `mysql` parses `/* ... */` comments:

- A single-quote, double-quote, or backtick character is taken to indicate the beginning of a quoted string or identifier, 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 `` `> ``. This problem was fixed in MySQL 4.1.1.

- A semicolon within the comment is taken to indicate the end of the current SQL statement and anything following it to indicate the beginning of the next statement. This problem was fixed in MySQL 4.0.13.

For affected versions of MySQL, these limitations apply both when you run `mysql` interactively and when you put commands in a file and use `mysql` in batch mode to process the file with `mysql < file_name`.

## 2.6 Treatment of Reserved Words in MySQL

A common problem stems from trying to use an identifier such as a table or column name that is the name of a built-in MySQL data type or function, such as `TIMESTAMP` or `GROUP`. You're allowed to do this (for example, `ABS` is allowed as a column name). However, by default, no whitespace is allowed in function invocations between the function name and the following '`(`' character. This requirement allows a function call to be distinguished from a reference to a column name.

A side effect of this behavior is that omitting a space in some contexts causes an identifier to be interpreted as a function name. For example, this statement is legal:

```
mysql> CREATE TABLE abs (val INT);
```

But omitting the space after `abs` causes a syntax error because the statement then appears to invoke the `ABS()` function:

```
mysql> CREATE TABLE abs(val INT);
```

If the server SQL mode includes the `IGNORE_SPACE` mode value, the server allows function invocations to have whitespace between a function name and the following '`(`' character. This causes function names to be treated as reserved words. As a result, identifiers that are the same as function names must be quoted as described in Section 2.2, "Database, Table, Index, Column, and Alias Names." The server SQL mode is controlled as described in Section 1.8.2, "Selecting SQL Modes."

The words in the following table are explicitly reserved in MySQL. Most of them are forbidden by standard SQL as column and/or table names (for example, `GROUP`). A few are reserved because MySQL needs them and (currently) uses a `yacc` parser. A reserved word can be used as an identifier by quoting it.

| | | |
|---|---|---|
| ADD | ALL | ALTER |
| ANALYZE | AND | AS |
| ASC | ASENSITIVE | AUTO_INCREMENT |
| BDB | BEFORE | BERKELEYDB |

| | | |
|---|---|---|
| BETWEEN | BIGINT | BINARY |
| BLOB | BOTH | BY |
| CALL | CASCADE | CASE |
| CHANGE | CHAR | CHARACTER |
| CHECK | COLLATE | COLUMN |
| COLUMNS | CONDITION | CONNECTION |
| CONSTRAINT | CONTINUE | CREATE |
| CROSS | CURRENT_DATE | CURRENT_TIME |
| CURRENT_TIMESTAMP | CURSOR | DATABASE |
| DATABASES | DAY_HOUR | DAY_MICROSECOND |
| DAY_MINUTE | DAY_SECOND | DEC |
| DECIMAL | DECLARE | DEFAULT |
| DELAYED | DELETE | DESC |
| DESCRIBE | DETERMINISTIC | DISTINCT |
| DISTINCTROW | DIV | DOUBLE |
| DROP | ELSE | ELSEIF |
| ENCLOSED | ESCAPED | EXISTS |
| EXIT | EXPLAIN | FALSE |
| FETCH | FIELDS | FLOAT |
| FOR | FORCE | FOREIGN |
| FOUND | FRAC_SECOND | FROM |
| FULLTEXT | GRANT | GROUP |
| HAVING | HIGH_PRIORITY | HOUR_MICROSECOND |
| HOUR_MINUTE | HOUR_SECOND | IF |
| IGNORE | IN | INDEX |
| INFILE | INNER | INNODB |
| INOUT | INSENSITIVE | INSERT |
| INT | INTEGER | INTERVAL |
| INTO | IO_THREAD | IS |
| ITERATE | JOIN | KEY |
| KEYS | KILL | LEADING |
| LEAVE | LEFT | LIKE |
| LIMIT | LINES | LOAD |
| LOCALTIME | LOCALTIMESTAMP | LOCK |
| LONG | LONGBLOB | LONGTEXT |
| LOOP | LOW_PRIORITY | MASTER_SERVER_ID |
| MATCH | MEDIUMBLOB | MEDIUMINT |
| MEDIUMTEXT | MIDDLEINT | MINUTE_MICROSECOND |
| MINUTE_SECOND | MOD | NATURAL |
| NOT | NO_WRITE_TO_BINLOG | NULL |

| | | |
|---|---|---|
| NUMERIC | ON | OPTIMIZE |
| OPTION | OPTIONALLY | OR |
| ORDER | OUT | OUTER |
| OUTFILE | PRECISION | PRIMARY |
| PRIVILEGES | PROCEDURE | PURGE |
| READ | REAL | REFERENCES |
| REGEXP | RENAME | REPEAT |
| REPLACE | REQUIRE | RESTRICT |
| RETURN | REVOKE | RIGHT |
| RLIKE | SECOND_MICROSECOND | SELECT |
| SENSITIVE | SEPARATOR | SET |
| SHOW | SMALLINT | SOME |
| SONAME | SPATIAL | SPECIFIC |
| SQL | SQLEXCEPTION | SQLSTATE |
| SQLWARNING | SQL_BIG_RESULT | SQL_CALC_FOUND_ROWS |
| SQL_SMALL_RESULT | SQL_TSI_DAY | SQL_TSI_FRAC_SECOND |
| SQL_TSI_HOUR | SQL_TSI_MINUTE | SQL_TSI_MONTH |
| SQL_TSI_QUARTER | SQL_TSI_SECOND | SQL_TSI_WEEK |
| SQL_TSI_YEAR | SSL | STARTING |
| STRAIGHT_JOIN | STRIPED | TABLE |
| TABLES | TERMINATED | THEN |
| TIMESTAMPADD | TIMESTAMPDIFF | TINYBLOB |
| TINYINT | TINYTEXT | TO |
| TRAILING | TRUE | UNDO |
| UNION | UNIQUE | UNLOCK |
| UNSIGNED | UPDATE | USAGE |
| USE | USER_RESOURCES | USING |
| UTC_DATE | UTC_TIME | UTC_TIMESTAMP |
| VALUES | VARBINARY | VARCHAR |
| VARCHARACTER | VARYING | WHEN |
| WHERE | WHILE | WITH |
| WRITE | XOR | YEAR_MONTH |
| ZEROFILL | | |

The following keywords are allowed by MySQL as column/table names. This is because they are very natural names and a lot of people have already used them.

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

# 3 Character Set Support

Improved support for character set handling was added to MySQL in Version 4.1. The features described here are as implemented in MySQL 4.1.1. (MySQL 4.1.0 has some but not all of these features, and some of them are implemented differently.)

This chapter discusses the following topics:

- What are character sets and collations?
- The multiple-level default system
- New syntax in MySQL 4.1
- Affected functions and operations
- Unicode support
- The meaning of each individual character set and collation

Character set support currently is included in the `MySISAM`, `MEMORY` (`HEAP`), and (as of MySQL 4.1.2) `InnoDB` storage engines. The `ISAM` storage engine does not include character set support; there are no plans to change this, because `ISAM` is deprecated.

## 3.1 Character Sets and Collations in General

A **character set** is a set of symbols and encodings. A **collation** is a set of rules for comparing characters in a character set. Let's make the distinction clear with an example of an imaginary character set.

Suppose that we have an alphabet with four letters: '`A`', '`B`', '`a`', '`b`'. We give each letter a number: '`A`' = 0, '`B`' = 1, '`a`' = 2, '`c`' = 3. The letter '`A`' is a symbol, the number 0 is the **encoding** for '`A`', and the combination of all four letters and their encodings is a **character set**.

Now, suppose that we want to compare two string values, '`A`' and '`B`'. The simplest way to do this is to look at the encodings: 0 for '`A`' and 1 for '`B`'. Because 0 is less than 1, we say '`A`' is less than '`B`'. Now, what we've just done is apply a collation to our character set. The collation is a set of rules (only one rule in this case): "compare the encodings." We call this simplest of all possible collations a **binary** collation.

But what if we want to say that the lowercase and uppercase letters are equivalent? Then we would have at least two rules: (1) treat the lowercase letters 'a' and 'b' as equivalent to 'A' and 'B'; (2) then compare the encodings. We call this a **case-insensitive** collation. It's a little more complex than a binary collation.

In real life, most character sets have many characters: not just 'A' and 'B' but whole alphabets, sometimes multiple alphabets or eastern writing systems with thousands of characters, along with many special symbols and punctuation marks. Also in real life, most collations have many rules: not just case insensitivity but also accent insensitivity (an "accent" is a mark attached to a character as in German 'ö') and multiple-character mappings (such as the rule that 'ö' = 'OE' in one of the two German collations).

MySQL 4.1 can do these things for you:

- Store strings using a variety of character sets
- Compare strings using a variety of collations
- Mix strings with different character sets or collations in the same server, the same database, or even the same table
- Allow specification of character set and collation at any level

In these respects, not only is MySQL 4.1 far more flexible than MySQL 4.0, it also is far ahead of other DBMSs. However, to use the new features effectively, you will need to learn what character sets and collations are available, how to change their defaults, and what the various string operators do with them.

## 3.2 Character Sets and Collations in MySQL

The MySQL server can support multiple character sets. To list the available character sets, use the SHOW CHARACTER SET statement:

```
mysql> SHOW CHARACTER SET;
+----------+-----------------------------+---------------------+
| Charset  | Description                 | Default collation   |
+----------+-----------------------------+---------------------+
| big5     | Big5 Traditional Chinese    | big5_chinese_ci     |
| dec8     | DEC West European           | dec8_swedish_ci     |
| cp850    | DOS West European           | cp850_general_ci    |
| hp8      | HP West European            | hp8_english_ci      |
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    |
| latin1   | ISO 8859-1 West European    | latin1_swedish_ci   |
| latin2   | ISO 8859-2 Central European | latin2_general_ci   |
...
```

The output actually includes another column that is not shown so that the example fits better on the page.

Any given character set always has at least one collation. It may have several collations.

To list the collations for a character set, use the `SHOW COLLATION` statement. For example, to see the collations for the `latin1` ("ISO-8859-1 West European") character set, use this statement to find those collation names that begin with `latin1`:

```
mysql> SHOW COLLATION LIKE 'latin1%';
+-------------------+---------+----+---------+----------+---------+
| Collation         | Charset | Id | Default | Compiled | Sortlen |
+-------------------+---------+----+---------+----------+---------+
| latin1_german1_ci | latin1  |  5 |         |          |       0 |
| latin1_swedish_ci | latin1  |  8 | Yes     | Yes      |       1 |
| latin1_danish_ci  | latin1  | 15 |         |          |       0 |
| latin1_german2_ci | latin1  | 31 |         | Yes      |       2 |
| latin1_bin        | latin1  | 47 |         | Yes      |       1 |
| latin1_general_ci | latin1  | 48 |         |          |       0 |
| latin1_general_cs | latin1  | 49 |         |          |       0 |
| latin1_spanish_ci | latin1  | 94 |         |          |       0 |
+-------------------+---------+----+---------+----------+---------+
```

The `latin1` collations have the following meanings:

| **Collation** | **Meaning** |
|---|---|
| `latin1_bin` | Binary according to `latin1` encoding |
| `latin1_danish_ci` | Danish/Norwegian |
| `latin1_general_ci` | Multilingual |
| `latin1_general_cs` | Multilingual, case sensitive |
| `latin1_german1_ci` | German DIN-1 |
| `latin1_german2_ci` | German DIN-2 |
| `latin1_spanish_ci` | Modern Spanish |
| `latin1_swedish_ci` | Swedish/Finnish |

Collations have these general characteristics:

- Two different character sets cannot have the same collation.
- Each character set has one collation that is the *default collation*. For example, the default collation for `latin1` is `latin1_swedish_ci`.
- There is a convention for collation names: They start with the name of the character set with which they are associated, they usually include a language name, and they end with `_ci` (case insensitive), `_cs` (case sensitive), `_bin` (binary), or `_uca` (Unicode Collation Algorithm, `http://www.unicode.org/reports/tr10/`).

## 3.3 Determining the Default Character Set and Collation

There are default settings for character sets and collations at four levels: server, database, table, and connection. The following description may appear complex, but it has been found in practice that multiple-level defaulting leads to natural and obvious results.

### 3.3.1 Server Character Set and Collation

The MySQL Server has a server character set and a server collation, which may not be null.

MySQL determines the server character set and server collation thus:

- According to the option settings in effect when the server starts
- According to the values set at runtime

At the server level, the decision is simple. The server character set and collation depend initially on the options that you use when you start `mysqld`. You can use `--default-character-set` for the character set, and along with it you can add `--default-collation` for the collation. If you don't specify a character set, that is the same as saying `--default-character-set=latin1`. If you specify only a character set (for example, `latin1`) but not a collation, that is the same as saying `--default-charset=latin1 --default-collation=latin1_swedish_ci` because `latin1_swedish_ci` is the default collation for `latin1`. Therefore, the following three commands all have the same effect:

```
shell> mysqld
shell> mysqld --default-character-set=latin1
shell> mysqld --default-character-set=latin1 \
           --default-collation=latin1_swedish_ci
```

One way to change the settings is by recompiling. If you want to change the default server character set and collation when building from sources, use: `--with-charset` and `--with-collation` as arguments for `configure`. For example:

```
shell> ./configure --with-charset=latin1
```

Or:

```
shell> ./configure --with-charset=latin1 \
           --with-collation=latin1_german1_ci
```

Both `mysqld` and `configure` verify that the character set/collation combination is valid. If not, each program displays an error message and terminates.

The current server character set and collation are available as the values of the `character_set_server` and `collation_server` system variables. These variables can be changed at runtime.

## 3.3.2 Database Character Set and Collation

Every database has a database character set and a database collation, which may not be null. The `CREATE DATABASE` and `ALTER DATABASE` statements have optional clauses for specifying the database character set and collation:

```
CREATE DATABASE db_name
    [[DEFAULT] CHARACTER SET charset_name]
    [[DEFAULT] COLLATE collation_name]

ALTER DATABASE db_name
    [[DEFAULT] CHARACTER SET charset_name]
    [[DEFAULT] COLLATE collation_name]
```

Example:

```
CREATE DATABASE db_name
    DEFAULT CHARACTER SET latin1 COLLATE latin1_swedish_ci;
```

MySQL chooses the database character set and database collation thus:

- If both `CHARACTER SET` *`X`* and `COLLATE` *`Y`* were specified, then character set *`X`* and collation *`Y`*.
- If `CHARACTER SET` *`X`* was specified without `COLLATE`, then character set *`X`* and its default collation.
- Otherwise, the server character set and server collation.

MySQL's `CREATE DATABASE ... DEFAULT CHARACTER SET ...` syntax is analogous to the standard SQL `CREATE SCHEMA ... CHARACTER SET ...` syntax. Because of this, it is possible to create databases with different character sets and collations on the same MySQL server.

The database character set and collation are used as default values if the table character set and collation are not specified in `CREATE TABLE` statements. They have no other purpose.

The character set and collation for the default database are available as the values of the `character_set_database` and `collation_database` system variables. The server sets these variables whenever the default database changes. If there is no default database, the variables have the same value as the corresponding server-level variables, `character_set_server` and `collation_server`.

## 3.3.3 Table Character Set and Collation

Every table has a table character set and a table collation, which may not be null. The CREATE TABLE and ALTER TABLE statements have optional clauses for specifying the table character set and collation:

```
CREATE TABLE tbl_name (column_list)
    [DEFAULT CHARACTER SET charset_name [COLLATE collation_name]]

ALTER TABLE tbl_name
    [DEFAULT CHARACTER SET charset_name] [COLLATE collation_name]
```

Example:

```
CREATE TABLE t1 ( ... )
    DEFAULT CHARACTER SET latin1 COLLATE latin1_danish_ci;
```

MySQL chooses the table character set and collation thus:

- If both CHARACTER SET *X* and COLLATE *Y* were specified, then character set *X* and collation *Y*.
- If CHARACTER SET *X* was specified without COLLATE, then character set *X* and its default collation.
- Otherwise, the database character set and collation.

The table character set and collation are used as default values if the column character set and collation are not specified in individual column definitions. The table character set and collation are MySQL extensions; there are no such things in standard SQL.

## 3.3.4 Column Character Set and Collation

Every "character" column (that is, a column of type CHAR, VARCHAR, or TEXT) has a column character set and a column collation, which may not be null. Column definition syntax has optional clauses for specifying the column character set and collation:

```
col_name {CHAR | VARCHAR | TEXT} (col_length)
    [CHARACTER SET charset_name [COLLATE collation_name]]
```

Example:

```
CREATE TABLE Table1
(
    column1 VARCHAR(5) CHARACTER SET latin1 COLLATE latin1_german1_ci
);
```

MySQL chooses the column character set and collation thus:

- If both `CHARACTER SET` *X* and `COLLATE` *Y* were specified, then character set *X* and collation *Y*.
- If `CHARACTER SET` *X* was specified without `COLLATE`, then character set *X* and its default collation.
- Otherwise, the table character set and collation.

The `CHARACTER SET` and `COLLATE` clauses are standard SQL.

## 3.3.5 Examples of Character Set and Collation Assignment

The following examples show how MySQL determines default character set and collation values.

### Example 1: Table + Column Definition

```
CREATE TABLE t1
(
    c1 CHAR(10) CHARACTER SET latin1 COLLATE latin1_german1_ci
) DEFAULT CHARACTER SET latin2 COLLATE latin2_bin;
```

Here we have a column with a `latin1` character set and a `latin1_german1_ci` collation. The definition is explicit, so that's straightforward. Notice that there's no problem storing a `latin1` column in a `latin2` table.

### Example 2: Table + Column Definition

```
CREATE TABLE t1
(
    c1 CHAR(10) CHARACTER SET latin1
) DEFAULT CHARACTER SET latin1 COLLATE latin1_danish_ci;
```

This time we have a column with a `latin1` character set and a default collation. Now, although it might seem natural, the default collation is not taken from the table level. Instead, because the default collation for `latin1` is always `latin1_swedish_ci`, column `c1` will have a collation of `latin1_swedish_ci` (not `latin1_danish_ci`).

### Example 3: Table + Column Definition

```
CREATE TABLE t1
(
    c1 CHAR(10)
) DEFAULT CHARACTER SET latin1 COLLATE latin1_danish_ci;
```

We have a column with a default character set and a default collation. In this circumstance, MySQL looks up to the table level for inspiration in determining the column character set and collation. So, the character set for column `c1` is `latin1` and its collation is `latin1_danish_ci`.

### Example 4: Database + Table + Column Definition

```
CREATE DATABASE d1
    DEFAULT CHARACTER SET latin2 COLLATE latin2_czech_ci;
USE d1;
CREATE TABLE t1
(
    c1 CHAR(10)
);
```

We create a column without specifying its character set and collation. We're also not specifying a character set and a collation at the table level. In this circumstance, MySQL looks up to the database level for inspiration. (The database's settings become the table's settings, and thereafter become the column's setting.) So, the character set for column `c1` is `latin2` and its collation is `latin2_czech_ci`.

## 3.3.6 Connection Character Sets and Collations

Several character set and collation system variables relate to a client's interaction with the server. Some of these have already been mentioned in earlier sections:

- The server character set and collation are available as the values of the `character_set_server` and `collation_server` variables.
- The character set and collation of the default database are available as the values of the `character_set_database` and `collation_database` variables.

Additional character set and collation variables are involved in handling traffic for the connection between a client and the server. Every client has connection-related character set and collation variables.

Consider what a "connection" is: It's what you make when you connect to the server. The client sends SQL statements, such as queries, over the connection to the server. The server sends responses, such as result sets, over the connection back to the client. This leads to several questions about character set and collation handling for client connections, each of which can be answered in terms of system variables:

- What character set is the query in when it leaves the client?

  The server takes the `character_set_client` variable to be the character set in which queries are sent by the client.
- What character set should the server translate a query to after receiving it?

  For this, `character_set_connection` and `collation_connection` are used by the server. It converts queries sent by the client from `character_set_client` to `character_set_connection` (except for string literals that have an introducer such as `_latin1` or `_utf8`). `collation_connection` is important for comparisons of literal strings. For comparisons of strings with column values, it does not matter because columns have a higher collation precedence.

- What character set should the server translate to before shipping result sets or error messages back to the client?

  The `character_set_results` variable indicates the character set in which the server returns query results to the client. This includes result data such as column values, and result metadata such as column names.

You can fine-tune the settings for these variables, or you can depend on the defaults (in which case, you can skip this section).

There are two statements that affect the connection character sets:

```
SET NAMES 'charset_name'
SET CHARACTER SET charset_name
```

`SET NAMES` indicates what is in the SQL statements that the client sends. Thus, `SET NAMES 'cp1251'` tells the server "future incoming messages from this client will be in character set `cp1251`." It also specifies the character set for results that the server sends back to the client. (For example, it indicates what character set column values will have if you use a `SELECT` statement.)

A `SET NAMES 'x'` statement is equivalent to these three statements:

```
mysql> SET character_set_client = x;
mysql> SET character_set_results = x;
mysql> SET character_set_connection = x;
```

Setting `character_set_connection` to *x* also sets `collation_connection` to the default `collation` for *x*.

`SET CHARACTER SET` is similar but sets the connection character set and collation to be those of the default database. A `SET CHARACTER SET x` statement is equivalent to these three statements:

```
mysql> SET character_set_client = x;
mysql> SET character_set_results = x;
mysql> SET collation_connection = @@collation_database;
```

When a client connects, it sends to the server the name of the character set that it wants to use. The server sets the `character_set_client`, `character_set_results`, and `character_set_connection` variables to that character set. (In effect, the server performs a `SET NAMES` operation using the character set.)

With the `mysql` client, it is not necessary to execute `SET NAMES` every time you start up if you want to use a character set different from the default. You can add the `--default-character-set` option setting to your `mysql` statement line, or in your option file. For example, the following option file setting changes the three character set variables set to `koi8r` each time you run `mysql`:

```
[mysql]
default-character-set=koi8r
```

Example: Suppose that `column1` is defined as `CHAR(5) CHARACTER SET latin2`. If you do not say `SET NAMES` or `SET CHARACTER SET`, then for `SELECT column1 FROM t`, the server will send back all the values for `column1` using the character set that the client specified when it connected. On the other hand, if you say `SET NAMES 'latin1'` or `SET CHARACTER SET latin1`, then just before sending results back, the server will convert the `latin2` values to `latin1`. Conversion may be lossy if there are characters that are not in both character sets.

If you do not want the server to perform any conversion, set `character_set_results` to `NULL`:

```
mysql> SET character_set_results = NULL;
```

## 3.3.7 Character String Literal Character Set and Collation

Every character string literal has a character set and a collation, which may not be null.

A character string literal may have an optional character set introducer and `COLLATE` clause:

```
[_charset_name]'string' [COLLATE collation_name]
```

Examples:

```
SELECT 'string';
SELECT _latin1'string';
SELECT _latin1'string' COLLATE latin1_danish_ci;
```

For the simple statement `SELECT 'string'`, the string has the character set and collation defined by the `character_set_connection` and `collation_connection` system variables.

The *`_charset_name`* expression is formally called an *introducer*. It tells the parser, "the string that is about to follow is in character set *`X`*." Because this has confused people in the past, we emphasize that an introducer does not cause any conversion, it is strictly a signal that does not change the string's value. An introducer is also legal before standard hex literal and numeric hex literal notation (`x'literal'` and `0xnnnn`), and before `?` (parameter substitution when using prepared statements within a programming language interface).

Examples:

```
SELECT _latin1 x'AABBCC';
SELECT _latin1 0xAABBCC;
SELECT _latin1 ?;
```

MySQL determines a literal's character set and collation thus:

- If both *`_X`* and `COLLATE` *`Y`* were specified, then character set *`X`* and collation *`Y`*
- If *`_X`* is specified but `COLLATE` is not specified, then character set *`X`* and its default collation
- Otherwise, the character set and collation given by the `character_set_connection` and `collation_connection` system variables

Examples:

- A string with `latin1` character set and `latin1_german1_ci` collation:

  ```
  SELECT _latin1'Müller' COLLATE latin1_german1_ci;
  ```

- A string with `latin1` character set and its default collation (that is, `latin1_swedish_ci`):

  ```
  SELECT _latin1'Müller';
  ```

- A string with the connection default character set and collation:

  ```
  SELECT 'Müller';
  ```

Character set introducers and the `COLLATE` clause are implemented according to standard SQL specifications.

## 3.3.8 Using `COLLATE` in SQL Statements

With the `COLLATE` clause, you can override whatever the default collation is for a comparison. `COLLATE` may be used in various parts of SQL statements. Here are some examples:

- With `ORDER BY`:

  ```
  SELECT k
  FROM t1
  ORDER BY k COLLATE latin1_german2_ci;
  ```

- With `AS`:

  ```
  SELECT k COLLATE latin1_german2_ci AS k1
  FROM t1
  ORDER BY k1;
  ```

- With `GROUP BY`:

  ```
  SELECT k
  FROM t1
  GROUP BY k COLLATE latin1_german2_ci;
  ```

- With aggregate functions:

  ```
  SELECT MAX(k COLLATE latin1_german2_ci)
  FROM t1;
  ```

- With `DISTINCT`:

  ```
  SELECT DISTINCT k COLLATE latin1_german2_ci
  FROM t1;
  ```

- With `WHERE`:

  ```
  SELECT *
  FROM t1
  WHERE _latin1 'Müller' COLLATE latin1_german2_ci = k;
  ```

- With HAVING:

  ```
  SELECT k
  FROM t1
  GROUP BY k
  HAVING k = _latin1 'Müller' COLLATE latin1_german2_ci;
  ```

### 3.3.9 COLLATE Clause Precedence

The COLLATE clause has high precedence (higher than ||), so the following two expressions are equivalent:

```
x || y COLLATE z
x || (y COLLATE z)
```

### 3.3.10 BINARY Operator

The BINARY operator is a shorthand for a COLLATE clause. BINARY '*x*' is equivalent to '*x*' COLLATE *y*, where *y* is the name of the binary collation for the character set of '*x*'. Every character set has a binary collation. For example, the binary collation for the latin1 character set is latin1_bin, so if the column a is of character set latin1, the following two statements have the same effect:

```
SELECT * FROM t1 ORDER BY BINARY a;
SELECT * FROM t1 ORDER BY a COLLATE latin1_bin;
```

### 3.3.11 Some Special Cases Where the Collation Determination Is Tricky

In the great majority of queries, it is obvious what collation MySQL uses to resolve a comparison operation. For example, in the following cases, it should be clear that the collation will be "the column collation of column x":

```
SELECT x FROM T ORDER BY x;
SELECT x FROM T WHERE x = x;
SELECT DISTINCT x FROM T;
```

However, when multiple operands are involved, there can be ambiguity. For example:

```
SELECT x FROM T WHERE x = 'Y';
```

Should this query use the collation of the column x, or of the string literal 'Y'?

Standard SQL resolves such questions using what used to be called "coercibility" rules. The essence is: Because x and 'Y' both have collations, whose collation takes precedence? It's complex, but the following rules take care of most situations:

- An explicit COLLATE clause has a coercibility of 0. (Not coercible at all.)
- A concatenation of two strings with different collations has a coercibility of 1.

- A column's collation has a coercibility of 2.
- A literal's collation has a coercibility of 3.

Those rules resolve ambiguities thus:

- Use the collation with the lowest coercibility value.
- If both sides have the same coercibility, then it is an error if the collations aren't the same.

Examples:

| | |
|---|---|
| `column1 = 'A'` | Use collation of `column1` |
| `column1 = 'A' COLLATE` *`x`* | Use collation of `'A'` |
| `column1 COLLATE` *`x`* `= 'A' COLLATE` *`y`* | Error |

The `COERCIBILITY()` function can be used to determine the coercibility of a string expression:

```
mysql> SELECT COERCIBILITY('A' COLLATE latin1_swedish_ci);
        -> 0
mysql> SELECT COERCIBILITY('A');
        -> 3
```

See Section 5.8.3, "Information Functions."

## 3.3.12 Collations Must Be for the Right Character Set

Recall that each character set has one or more collations, and each collation is associated with one and only one character set. Therefore, the following statement causes an error message because the `latin2_bin` collation is not legal with the `latin1` character set:

```
mysql> SELECT _latin1 'x' COLLATE latin2_bin;
ERROR 1251: COLLATION 'latin2_bin' is not valid
for CHARACTER SET 'latin1'
```

In some cases, expressions that worked before MySQL 4.1 fail as of MySQL 4.1 if you do not take character set and collation into account. For example, before 4.1, this statement works as is:

```
mysql> SELECT SUBSTRING_INDEX(USER(),'@',1);
+-------------------------------+
| SUBSTRING_INDEX(USER(),'@',1) |
+-------------------------------+
| root                          |
+-------------------------------+
```

After an upgrade to MySQL 4.1, the statement fails:

```
mysql> SELECT SUBSTRING_INDEX(USER(),'@',1);
ERROR 1267 (HY000): Illegal mix of collations
(utf8_general_ci,IMPLICIT) and (latin1_swedish_ci,COERCIBLE)
for operation 'substr_index'
```

The reason this occurs is that usernames are stored using UTF8 (see Section 3.6, "UTF8 for Metadata"). As a result, the `USER()` function and the literal string `'@'` have different character sets (and thus different collations):

```
mysql> SELECT COLLATION(USER()), COLLATION('@');
+-------------------+-------------------+
| COLLATION(USER()) | COLLATION('@')    |
+-------------------+-------------------+
| utf8_general_ci   | latin1_swedish_ci |
+-------------------+-------------------+
```

One way to deal with this is to tell MySQL to interpret the literal string as `utf8`:

```
mysql> SELECT SUBSTRING_INDEX(USER(),_utf8'@',1);
+------------------------------------+
| SUBSTRING_INDEX(USER(),_utf8'@',1) |
+------------------------------------+
| root                               |
+------------------------------------+
```

Another way is to change the connection character set and collation to `utf8`. You can do that with `SET NAMES 'utf8'` or by setting the `character_set_connection` and `collation_connection` system variables directly.

## 3.3.13 An Example of the Effect of Collation

Suppose that column `X` in table `T` has these `latin1` column values:

```
Muffler
Müller
MX Systems
MySQL
```

And suppose that the column values are retrieved using the following statement:

```
SELECT X FROM T ORDER BY X COLLATE collation_name;
```

The resulting order of the values for different collations is shown in this table:

| `latin1_swedish_ci` | `latin1_german1_ci` | `latin1_german2_ci` |
|---|---|---|
| Muffler | Muffler | Müller |
| MX Systems | Müller | Muffler |
| Müller | MX Systems | MX Systems |
| MySQL | MySQL | MySQL |

The table is an example that shows what the effect would be if we used different collations in an `ORDER BY` clause. The character that causes the different sort orders in this example is the U with two dots over it, which the Germans call U-umlaut, but we'll call it U-dieresis.

- The first column shows the result of the `SELECT` using the Swedish/Finnish collating rule, which says that U-dieresis sorts with Y.
- The second column shows the result of the `SELECT` using the German DIN-1 rule, which says that U-dieresis sorts with U.
- The third column shows the result of the `SELECT` using the German DIN-2 rule, which says that U-dieresis sorts with UE.

Three different collations, three different results. That's what MySQL is here to handle. By using the appropriate collation, you can choose the sort order you want.

## 3.4 Operations Affected by Character Set Support

This section describes operations that take character set information into account as of MySQL 4.1.

### 3.4.1 Result Strings

MySQL has many operators and functions that return a string. This section answers the question: What is the character set and collation of such a string?

For simple functions that take string input and return a string result as output, the output's character set and collation are the same as those of the principal input value. For example, `UPPER(X)` returns a string whose character string and collation are the same as that of *X*. The same applies for `INSTR()`, `LCASE()`, `LOWER()`, `LTRIM()`, `MID()`, `REPEAT()`, `REPLACE()`, `REVERSE()`, `RIGHT()`, `RPAD()`, `RTRIM()`, `SOUNDEX()`, `SUBSTRING()`, `TRIM()`, `UCASE()`, and `UPPER()`. (Also note: The `REPLACE()` function, unlike all other functions, ignores the collation of the string input and performs a case-insensitive comparison every time.)

For operations that combine multiple string inputs and return a single string output, the "aggregation rules" of standard SQL apply:

- If an explicit `COLLATE` *`X`* occurs, then use *`X`*
- If an explicit `COLLATE` *`X`* and `COLLATE` *`Y`* occur, then error
- Otherwise, if all collations are *`X`*, then use *`X`*
- Otherwise, the result has no collation

For example, with `CASE ... WHEN a THEN b WHEN b THEN c COLLATE` *`X`* `END`, the resultant collation is *`X`*. The same applies for `CASE`, `UNION`, `||`, `CONCAT()`, `ELT()`, `GREATEST()`, `IF()`, and `LEAST()`.

For operations that convert to character data, the character set and collation of the strings that result from the operations are defined by the `character_set_connection` and `collation_connection` system variables. This applies for `CAST()`, `CHAR()`, `CONV()`, `FORMAT()`, `HEX()`, and `SPACE()`.

## 3.4.2 CONVERT()

`CONVERT()` provides a way to convert data between different character sets. The syntax is:

```
CONVERT(expr USING transcoding_name)
```

In MySQL, transcoding names are the same as the corresponding character set names.

Examples:

```
SELECT CONVERT(_latin1'Müller' USING utf8);
INSERT INTO utf8table (utf8column)
    SELECT CONVERT(latin1field USING utf8) FROM latin1table;
```

`CONVERT(... USING ...)` is implemented according to the standard SQL specification.

## 3.4.3 CAST()

You may also use `CAST()` to convert a string to a different character set. The syntax is:

```
CAST(character_string AS character_data_type CHARACTER SET charset_name)
```

Example:

```
SELECT CAST(_latin1'test' AS CHAR CHARACTER SET utf8);
```

If you use `CAST()` without specifying `CHARACTER SET`, the resulting character set and collation are defined by the `character_set_connection` and `collation_connection` system variables. If you use `CAST()` with `CHARACTER SET` *`X`*, then the resulting character set and collation are *`X`* and the default collation of *`X`*.

You may not use a `COLLATE` clause inside a `CAST()`, but you may use it outside. That is, `CAST(... COLLATE ...)` is illegal, but `CAST(...) COLLATE ...` is legal.

Example:

```
SELECT CAST(_latin1'test' AS CHAR CHARACTER SET utf8) COLLATE utf8_bin;
```

### 3.4.4 SHOW Statements

Several `SHOW` statements are new or modified in MySQL 4.1 to provide additional character set information. `SHOW CHARACTER SET`, `SHOW COLLATION`, and `SHOW CREATE DATABASE` are new. `SHOW CREATE TABLE` and `SHOW COLUMNS` are modified.

The `SHOW CHARACTER SET` command shows all available character sets. It takes an optional `LIKE` clause that indicates which character set names to match. For example:

```
mysql> SHOW CHARACTER SET LIKE 'latin%';
+---------+-----------------------------+-------------------+--------+
| Charset | Description                 | Default collation | Maxlen |
+---------+-----------------------------+-------------------+--------+
| latin1  | ISO 8859-1 West European    | latin1_swedish_ci |      1 |
| latin2  | ISO 8859-2 Central European | latin2_general_ci |      1 |
| latin5  | ISO 8859-9 Turkish          | latin5_turkish_ci |      1 |
| latin7  | ISO 8859-13 Baltic          | latin7_general_ci |      1 |
+---------+-----------------------------+-------------------+--------+
```

See Section 6.5.3.2, "`SHOW CHARACTER SET` Syntax."

The output from `SHOW COLLATION` includes all available character sets. It takes an optional `LIKE` clause that indicates which collation names to match. For example:

```
mysql> SHOW COLLATION LIKE 'latin1%';
+-------------------+---------+----+---------+----------+---------+
| Collation         | Charset | Id | Default | Compiled | Sortlen |
+-------------------+---------+----+---------+----------+---------+
| latin1_german1_ci | latin1  |  5 |         |          |       0 |
| latin1_swedish_ci | latin1  |  8 | Yes     | Yes      |       0 |
| latin1_danish_ci  | latin1  | 15 |         |          |       0 |
| latin1_german2_ci | latin1  | 31 |         | Yes      |       2 |
| latin1_bin        | latin1  | 47 |         | Yes      |       0 |
| latin1_general_ci | latin1  | 48 |         |          |       0 |
| latin1_general_cs | latin1  | 49 |         |          |       0 |
| latin1_spanish_ci | latin1  | 94 |         |          |       0 |
+-------------------+---------+----+---------+----------+---------+
```

See Section 6.5.3.3, "SHOW COLLATION Syntax."

SHOW CREATE DATABASE displays the CREATE DATABASE statement that will create a given database. The result includes all database options. DEFAULT CHARACTER SET and COLLATE are supported. All database options are stored in a text file named db.opt that can be found in the database directory.

```
mysql> SHOW CREATE DATABASE a\G
*************************** 1. row ***************************
       Database: a
Create Database: CREATE DATABASE `a`
                 /*!40100 DEFAULT CHARACTER SET macce */
```

See Section 6.5.3.5, "SHOW CREATE DATABASE Syntax."

SHOW CREATE TABLE is similar, but displays the CREATE TABLE statement to create a given table. The column definitions now indicate any character set specifications, and the table options include character set information.

See Section 6.5.3.6, "SHOW CREATE TABLE Syntax."

The SHOW COLUMNS statement displays the collations of a table's columns when invoked as SHOW FULL COLUMNS. Columns with CHAR, VARCHAR, or TEXT data types have non-NULL collations. Numeric and other non-character types have NULL collations. For example:

```
mysql> SHOW FULL COLUMNS FROM t;
+-------+---------+------------+------+-----+---------+-------+
| Field | Type    | Collation  | Null | Key | Default | Extra |
+-------+---------+------------+------+-----+---------+-------+
| a     | char(1) | latin1_bin | YES  |     | NULL    |       |
| b     | int(11) | NULL       | YES  |     | NULL    |       |
+-------+---------+------------+------+-----+---------+-------+
```

The character set is not part of the display. (The character set name is implied by the collation name.)

See Section 6.5.3.4, "SHOW COLUMNS Syntax."

## 3.5 Unicode Support

As of MySQL version 4.1, there are two new character sets for storing Unicode data:

- ucs2, the UCS-2 Unicode character set.
- utf8, the UTF8 encoding of the Unicode character set.

In UCS-2 (binary Unicode representation), every character is represented by a two-byte Unicode code with the most significant byte first. For example: "LATIN CAPITAL LETTER A" has the code 0x0041 and it's stored as a two-byte sequence: 0x00 0x41. "CYRILLIC SMALL LETTER YERU" (Unicode 0x044B) is stored as a two-byte sequence: 0x04 0x4B. For Unicode characters and their codes, please refer to the Unicode Home Page (`http://www.unicode.org/`).

A temporary restriction is that UCS-2 cannot yet be used as a client character set. That means that `SET NAMES 'ucs2'` will not work.

The UTF8 character set (transform Unicode representation) is an alternative way to store Unicode data. It is implemented according to RFC2279. The idea of the UTF8 character set is that various Unicode characters fit into byte sequences of different lengths:

- Basic Latin letters, digits, and punctuation signs use one byte.
- Most European and Middle East script letters fit into a two-byte sequence: extended Latin letters (with tilde, macron, acute, grave, and other accents), Cyrillic, Greek, Armenian, Hebrew, Arabic, Syriac, and others.
- Korean, Chinese, and Japanese ideographs use three-byte sequences.

Currently, MySQL UTF8 support does not include four-byte sequences.

Tip: To save space with UTF8, use `VARCHAR` instead of `CHAR`. Otherwise, MySQL has to reserve 30 bytes for a `CHAR(10) CHARACTER SET utf8` column, because that's the maximum possible length.

## 3.6 UTF8 for Metadata

The metadata is the data about the data. Anything that describes the database, as opposed to being the contents of the database, is metadata. Thus column names, database names, user-names, version names, and most of the string results from `SHOW` are metadata.

Representation of metadata must satisfy these requirements:

- All metadata must be in the same character set. Otherwise, `SHOW` wouldn't work properly because different rows in the same column would be in different character sets.
- Metadata must include all characters in all languages. Otherwise, users wouldn't be able to name columns and tables in their own languages.

In order to satisfy both requirements, MySQL stores metadata in a Unicode character set, namely UTF8. This will not cause any disruption if you never use accented characters. But if you do, you should be aware that metadata is in UTF8.

This means that the `USER()`, `CURRENT_USER()`, and `VERSION()` functions will have the UTF8 character set by default. So will any synonyms, such as the `SESSION_USER()` and `SYSTEM_USER()` synonyms for `USER()`.

The server sets the `character_set_system` system variable to the name of the metadata character set:

```
mysql> SHOW VARIABLES LIKE 'character_set_system';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| character_set_system | utf8  |
+----------------------+-------+
```

Storage of metadata using Unicode does *not* mean that the headers of columns and the results of `DESCRIBE` functions will be in the `character_set_system` character set by default. When you say `SELECT column1 FROM t`, the name `column1` itself will be returned from the server to the client in the character set as determined by the `SET NAMES` statement. More specifically, the character set used is determined by the value of the `character_set_results` system variable. If this variable is set to `NULL`, no conversion is performed and the server returns metadata using its original character set (the set indicated by `character_set_system`).

If you want the server to pass metadata results back in a non-UTF8 character set, then use `SET NAMES` to force the server to perform character set conversion (see Section 3.3.6, “Connection Character Sets and Collations”), or else set the client to do the conversion. It is always more efficient to set the client to do the conversion, but this option will not be available for many clients until late in the MySQL 4.x product cycle.

If you are just using, for example, the `USER()` function for comparison or assignment within a single statement, don’t worry. MySQL will do some automatic conversion for you.

```
SELECT * FROM Table1 WHERE USER() = latin1_column;
```

This will work because the contents of `latin1_column` are automatically converted to UTF8 before the comparison.

```
INSERT INTO Table1 (latin1_column) SELECT USER();
```

This will work because the contents of `USER()` are automatically converted to `latin1` before the assignment. Automatic conversion is not fully implemented yet, but should work correctly in a later version.

Although automatic conversion is not in the SQL standard, the SQL standard document does say that every character set is (in terms of supported characters) a “subset” of Unicode. Since it is a well-known principle that “what applies to a superset can apply to a subset,” we believe that a collation for Unicode can apply for comparisons with non-Unicode strings.

## 3.7 Compatibility with Other DBMSs

For MaxDB compatibility these two statements are the same:

```
CREATE TABLE t1 (f1 CHAR(n) UNICODE);
CREATE TABLE t1 (f1 CHAR(n) CHARACTER SET ucs2);
```

## 3.8 New Character Set Configuration File Format

In MySQL 4.1, character set configuration is stored in XML files, one file per character set. In previous versions, this information was stored in `.conf` files.

## 3.9 National Character Set

Before MySQL 4.1, `NCHAR` and `CHAR` were synonymous. ANSI defines `NCHAR` or `NATIONAL CHAR` as a way to indicate that a `CHAR` column should use some predefined character set. MySQL 4.1 and up uses `utf8` as that predefined character set. For example, these column type declarations are equivalent:

```
CHAR(10) CHARACTER SET utf8
NATIONAL CHARACTER(10)
NCHAR(10)
```

As are these:

```
VARCHAR(10) CHARACTER SET utf8
NATIONAL VARCHAR(10)
NCHAR VARCHAR(10)
NATIONAL CHARACTER VARYING(10)
NATIONAL CHAR VARYING(10)
```

You can use `N'`*`literal`*`'` to create a string in the national character set. These two statements are equivalent:

```
SELECT N'some text';
SELECT _utf8'some text';
```

## 3.10 Upgrading Character Sets from MySQL 4.0

Now, what about upgrading from older versions of MySQL? MySQL 4.1 is almost upward compatible with MySQL 4.0 and earlier for the simple reason that almost all the features are new, so there’s nothing in earlier versions to conflict with. However, there are some differences and a few things to be aware of.

Most important: The "MySQL 4.0 character set" has the properties of both "MySQL 4.1 character sets" and "MySQL 4.1 collations." You will have to unlearn this. Henceforth, we will not bundle character set/collation properties in the same conglomerate object.

There is a special treatment of national character sets in MySQL 4.1. NCHAR is not the same as CHAR, and N'...' literals are not the same as '...' literals.

Finally, there is a different file format for storing information about character sets and collations. Make sure that you have reinstalled the /share/mysql/charsets/ directory containing the new configuration files.

If you want to start mysqld from a 4.1.x distribution with data created by MySQL 4.0, you should start the server with the same character set and collation. In this case you won't need to reindex your data.

There are two ways to do so:

```
shell> ./configure --with-charset=...  --with-collation=...
shell> ./mysqld --default-character-set=... --default-collation=...
```

If you used mysqld with, for example, the MySQL 4.0 danish character set, you should now use the latin1 character set and the latin1_danish_ci collation:

```
shell> ./configure --with-charset=latin1 \
           --with-collation=latin1_danish_ci
shell> ./mysqld --default-character-set=latin1 \
           --default-collation=latin1_danish_ci
```

Use the table shown in Section 3.10.1, "4.0 Character Sets and Corresponding 4.1 Character Set/Collation Pairs," to find old 4.0 character set names and their 4.1 character set/collation pair equivalents.

If you have non-latin1 data stored in a 4.0 latin1 table and want to convert the table column definitions to reflect the actual character set of the data, use the instructions in Section 3.10.2, "Converting 4.0 Character Columns to 4.1 Format."

## 3.10.1 4.0 Character Sets and Corresponding 4.1 Character Set/Collation Pairs

| ID | 4.0 Character Set | 4.1 Character Set | 4.1 Collation |
|---|---|---|---|
| 1 | big5 | big5 | big5_chinese_ci |
| 2 | czech | latin2 | latin2_czech_ci |
| 3 | dec8 | dec8 | dec8_swedish_ci |
| 4 | dos | cp850 | cp850_general_ci |
| 5 | german1 | latin1 | latin1_german1_ci |
| 6 | hp8 | hp8 | hp8_english_ci |
| 7 | koi8_ru | koi8r | koi8r_general_ci |

| ID | 4.0 Character Set | 4.1 Character Set | 4.1 Collation |
|---|---|---|---|
| 8 | latin1 | latin1 | latin1_swedish_ci |
| 9 | latin2 | latin2 | latin2_general_ci |
| 10 | swe7 | swe7 | swe7_swedish_ci |
| 11 | usa7 | ascii | ascii_general_ci |
| 12 | ujis | ujis | ujis_japanese_ci |
| 13 | sjis | sjis | sjis_japanese_ci |
| 14 | cp1251 | cp1251 | cp1251_bulgarian_ci |
| 15 | danish | latin1 | latin1_danish_ci |
| 16 | hebrew | hebrew | hebrew_general_ci |
| 17 | win1251 | (removed) | (removed) |
| 18 | tis620 | tis620 | tis620_thai_ci |
| 19 | euc_kr | euckr | euckr_korean_ci |
| 20 | estonia | latin7 | latin7_estonian_ci |
| 21 | hungarian | latin2 | latin2_hungarian_ci |
| 22 | koi8_ukr | koi8u | koi8u_ukrainian_ci |
| 23 | win1251ukr | cp1251 | cp1251_ukrainian_ci |
| 24 | gb2312 | gb2312 | gb2312_chinese_ci |
| 25 | greek | greek | greek_general_ci |
| 26 | win1250 | cp1250 | cp1250_general_ci |
| 27 | croat | latin2 | latin2_croatian_ci |
| 28 | gbk | gbk | gbk_chinese_ci |
| 29 | cp1257 | cp1257 | cp1257_lithuanian_ci |
| 30 | latin5 | latin5 | latin5_turkish_ci |
| 31 | latin1_de | latin1 | latin1_german2_ci |

### 3.10.2 Converting 4.0 Character Columns to 4.1 Format

Normally, the server runs using the `latin1` character set by default. If you have been storing column data that actually is in some other character set that the 4.1 server now supports directly, you can convert the column. However, you should avoid trying to convert directly from `latin1` to the "real" character set. This may result in data loss. Instead, convert the column to a binary column type, and then from the binary type to a non-binary type with the desired character set. Conversion to and from binary involves no attempt at character value conversion and preserves your data intact. For example, suppose that you have a 4.0 table with three columns that are used to store values represented in `latin1`, `latin2`, and `utf8`:

```
CREATE TABLE t
(
    latin1_col CHAR(50),
    latin2_col CHAR(100),
    utf8_col CHAR(150)
);
```

After upgrading to MySQL 4.1, you want to convert this table to leave `latin1_col` alone but change the `latin2_col` and `utf8_col` columns to have character sets of `latin2` and `utf8`. First, back up your table, then convert the columns as follows:

```
ALTER TABLE t MODIFY latin2_col BINARY(100);
ALTER TABLE t MODIFY utf8_col BINARY(150);
ALTER TABLE t MODIFY latin2_col CHAR(100) CHARACTER SET latin2;
ALTER TABLE t MODIFY utf8_col CHAR(150) CHARACTER SET utf8;
```

The first two statements "remove" the character set information from the `latin2_col` and `utf8_col` columns. The second two statements assign the proper character sets to the two columns.

If you like, you can combine the to-binary conversions and from-binary conversions into single statements:

```
ALTER TABLE t
    MODIFY latin2_col BINARY(100),
    MODIFY utf8_col BINARY(150);
ALTER TABLE t
    MODIFY latin2_col CHAR(100) CHARACTER SET latin2,
    MODIFY utf8_col CHAR(150) CHARACTER SET utf8;
```

## 3.11 Character Sets and Collations That MySQL Supports

Here is an annotated list of character sets and collations that MySQL supports. Because options and installation settings differ, some sites might not have all items listed, and some sites might have items not listed.

MySQL supports 70+ collations for 30+ character sets. The character sets and their default collations are displayed by the `SHOW CHARACTER SET` statement. (The output actually includes another column that is not shown so that the example fits better on the page.)

```
mysql> SHOW CHARACTER SET;
+----------+-----------------------------+---------------------+
| Charset  | Description                 | Default collation   |
+----------+-----------------------------+---------------------+
| big5     | Big5 Traditional Chinese    | big5_chinese_ci     |
| dec8     | DEC West European           | dec8_swedish_ci     |
| cp850    | DOS West European           | cp850_general_ci    |
| hp8      | HP West European            | hp8_english_ci      |
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    |
| latin1   | ISO 8859-1 West European    | latin1_swedish_ci   |
| latin2   | ISO 8859-2 Central European | latin2_general_ci   |
| swe7     | 7bit Swedish                | swe7_swedish_ci     |
| ascii    | US ASCII                    | ascii_general_ci    |
| ujis     | EUC-JP Japanese             | ujis_japanese_ci    |
| sjis     | Shift-JIS Japanese          | sjis_japanese_ci    |
| cp1251   | Windows Cyrillic            | cp1251_bulgarian_ci |
| hebrew   | ISO 8859-8 Hebrew           | hebrew_general_ci   |
| tis620   | TIS620 Thai                 | tis620_thai_ci      |
| euckr    | EUC-KR Korean               | euckr_korean_ci     |
| koi8u    | KOI8-U Ukrainian            | koi8u_general_ci    |
| gb2312   | GB2312 Simplified Chinese   | gb2312_chinese_ci   |
| greek    | ISO 8859-7 Greek            | greek_general_ci    |
| cp1250   | Windows Central European    | cp1250_general_ci   |
| gbk      | GBK Simplified Chinese      | gbk_chinese_ci      |
| latin5   | ISO 8859-9 Turkish          | latin5_turkish_ci   |
| armscii8 | ARMSCII-8 Armenian          | armscii8_general_ci |
| utf8     | UTF-8 Unicode               | utf8_general_ci     |
| ucs2     | UCS-2 Unicode               | ucs2_general_ci     |
| cp866    | DOS Russian                 | cp866_general_ci    |
| keybcs2  | DOS Kamenicky Czech-Slovak  | keybcs2_general_ci  |
| macce    | Mac Central European        | macce_general_ci    |
| macroman | Mac West European           | macroman_general_ci |
| cp852    | DOS Central European        | cp852_general_ci    |
| latin7   | ISO 8859-13 Baltic          | latin7_general_ci   |
| cp1256   | Windows Arabic              | cp1256_general_ci   |
| cp1257   | Windows Baltic              | cp1257_general_ci   |
| binary   | Binary pseudo charset       | binary              |
| geostd8  | GEOSTD8 Georgian            | geostd8_general_ci  |
+----------+-----------------------------+---------------------+
```

## 3.11.1 Unicode Character Sets

MySQL has two Unicode character sets. You can store texts in about 650 languages using these character sets. We have not added a large number of collations for these two new sets yet, but that will be happening soon. Currently, they have default case-insensitive accent-insensitive collations, plus the binary collation.

Currently, the `ucs2_general_uca` collation has only partial support for the Unicode Collation Algorithm. Some characters are not supported yet.

- `ucs2` (UCS-2 Unicode) collations:
    - `ucs2_bin`
    - `ucs2_general_ci` (default)
    - `ucs2_general_uca`
- `utf8` (UTF-8 Unicode) collations:
    - `utf8_bin`
    - `utf8_general_ci` (default)

## 3.11.2 West European Character Sets

West European Character Sets cover most West European languages, such as French, Spanish, Catalan, Basque, Portuguese, Italian, Albanian, Dutch, German, Danish, Swedish, Norwegian, Finnish, Faroese, Icelandic, Irish, Scottish, and English.

- `ascii` (US ASCII) collations:
  - `ascii_bin`
  - `ascii_general_ci` (default)
- `cp850` (DOS West European) collations:
  - `cp850_bin`
  - `cp850_general_ci` (default)
- `dec8` (DEC West European) collations:
  - `dec8_bin`
  - `dec8_swedish_ci` (default)
- `hp8` (HP West European) collations:
  - `hp8_bin`
  - `hp8_english_ci` (default)
- `latin1` (ISO 8859-1 West European) collations:
  - `latin1_bin`
  - `latin1_danish_ci`
  - `latin1_general_ci`
  - `latin1_general_cs`
  - `latin1_german1_ci`
  - `latin1_german2_ci`
  - `latin1_spanish_ci`
  - `latin1_swedish_ci` (default)

  The `latin1` is the default character set. The `latin1_swedish_ci` collation is the default that probably is used by the majority of MySQL customers. It is constantly stated that this is based on the Swedish/Finnish collation rules, but you will find Swedes and Finns who disagree with that statement.

  The `latin1_german1_ci` and `latin1_german2_ci` collations are based on the DIN-1 and DIN-2 standards, where DIN stands for Deutsches Institut für Normung (that is, the German answer to ANSI). DIN-1 is called the dictionary collation and DIN-2 is called the phone-book collation.

  - `latin1_german1_ci` (dictionary) rules:

    `'Ä' = 'A', 'Ö' = 'O', 'Ü' = 'U', 'ß' = 's'`

    - latin1_german2_ci (phone-book) rules:

      'Ä' = 'AE', 'Ö' = 'OE', 'Ü' = 'UE', 'ß' = 'ss'

  In the latin1_spanish_ci collation, 'Ñ' (N-tilde) is a separate letter between 'N' and 'O'.

- macroman (Mac West European) collations:
    - macroman_bin
    - macroman_general_ci (default)
- swe7 (7-bit Swedish) collations:
    - swe7_bin
    - swe7_swedish_ci (default)

### 3.11.3 Central European Character Sets

We have some support for character sets used in the Czech Republic, Slovakia, Hungary, Romania, Slovenia, Croatia, and Poland.

- cp1250 (Windows Central European) collations:
    - cp1250_bin
    - cp1250_czech_ci
    - cp1250_general_ci (default)
- cp852 (DOS Central European) collations:
    - cp852_bin
    - cp852_general_ci (default)
- keybcs2 (DOS Kamenicky Czech-Slovak) collations:
    - keybcs2_bin
    - keybcs2_general_ci (default)
- latin2 (ISO 8859-2 Central European) collations:
    - latin2_bin
    - latin2_croatian_ci
    - latin2_czech_ci
    - latin2_general_ci (default)
    - latin2_hungarian_ci
- macce (Mac Central European) collations:
    - macce_bin
    - macce_general_ci (default)

## 3.11.4 South European and Middle East Character Sets

- `armscii8` (ARMSCII-8 Armenian) collations:
    - `armscii8_bin`
    - `armscii8_general_ci` (default)
- `cp1256` (Windows Arabic) collations:
    - `cp1256_bin`
    - `cp1256_general_ci` (default)
- `geostd8` (GEOSTD8 Georgian) collations:
    - `geostd8_bin`
    - `geostd8_general_ci` (default)
- `greek` (ISO 8859-7 Greek) collations:
    - `greek_bin`
    - `greek_general_ci` (default)
- `hebrew` (ISO 8859-8 Hebrew) collations:
    - `hebrew_bin`
    - `hebrew_general_ci` (default)
- `latin5` (ISO 8859-9 Turkish) collations:
    - `latin5_bin`
    - `latin5_turkish_ci` (default)

## 3.11.5 Baltic Character Sets

The Baltic character sets cover Estonian, Latvian, and Lithuanian languages. There are two Baltic character sets currently supported:

- `cp1257` (Windows Baltic) collations:
    - `cp1257_bin`
    - `cp1257_general_ci` (default)
    - `cp1257_lithuanian_ci`
- `latin7` (ISO 8859-13 Baltic) collations:
    - `latin7_bin`
    - `latin7_estonian_cs`
    - `latin7_general_ci` (default)
    - `latin7_general_cs`

## 3.11.6 Cyrillic Character Sets

Here are the Cyrillic character sets and collations for use with Belarusian, Bulgarian, Russian, and Ukrainian languages.

- `cp1251` (Windows Cyrillic) collations:
  - `cp1251_bin`
  - `cp1251_bulgarian_ci`
  - `cp1251_general_ci` (default)
  - `cp1251_general_cs`
  - `cp1251_ukrainian_ci`
- `cp866` (DOS Russian) collations:
  - `cp866_bin`
  - `cp866_general_ci` (default)
- `koi8r` (KOI8-R Relcom Russian) collations:
  - `koi8r_bin`
  - `koi8r_general_ci` (default)
- `koi8u` (KOI8-U Ukrainian) collations:
  - `koi8u_bin`
  - `koi8u_general_ci` (default)

## 3.11.7 Asian Character Sets

The Asian character sets that we support include Chinese, Japanese, Korean, and Thai. These can be complicated. For example, the Chinese sets must allow for thousands of different characters.

- `big5` (Big5 Traditional Chinese) collations:
  - `big5_bin`
  - `big5_chinese_ci` (default)
- `euckr` (EUC-KR Korean) collations:
  - `euckr_bin`
  - `euckr_korean_ci` (default)
- `gb2312` (GB2312 Simplified Chinese) collations:
  - `gb2312_bin`
  - `gb2312_chinese_ci` (default)

- `gbk` (GBK Simplified Chinese) collations:
  - `gbk_bin`
  - `gbk_chinese_ci` (default)
- `sjis` (Shift-JIS Japanese) collations:
  - `sjis_bin`
  - `sjis_japanese_ci` (default)
- `tis620` (TIS620 Thai) collations:
  - `tis620_bin`
  - `tis620_thai_ci` (default)
- `ujis` (EUC-JP Japanese) collations:
  - `ujis_bin`
  - `ujis_japanese_ci` (default)

# 4 Column Types

MySQL supports a number of column types in several categories: numeric types, date and time types, and string (character) types. This chapter first gives an overview of these column types, and then provides a more detailed description of the properties of the types in each category, and a summary of the column type storage requirements. 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.

MySQL versions 4.1 and up support extensions for handling spatial data. Information about spatial types is provided in Chapter 7, "Spatial Extensions in MySQL."

Several of the column type descriptions use these conventions:

- *`M`*

  Indicates the maximum display size. The maximum legal display size is 255.
- *`D`*

  Applies to floating-point and fixed-point types and indicates the number of digits following the decimal point. The maximum possible value is 30, but should be no greater than *`M`*–2.
- Square brackets ('`[`' and '`]`') indicate parts of type specifiers that are optional.

## 4.1 Column Type Overview

### 4.1.1 Overview of Numeric Types

A summary of the numeric column types follows. For additional information, see Section 4.2, "Numeric Types." Column storage requirements are given in Section 4.5, "Column Type Storage Requirements."

If you specify `ZEROFILL` for a numeric column, MySQL automatically adds the `UNSIGNED` attribute to the column.

**Warning:** You should be aware that when you use subtraction between integer values where one is of type `UNSIGNED`, the result will be unsigned! See Section 5.7, "Cast Functions."

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

  A very small integer. The signed range is `-128` to `127`. The unsigned range is `0` to `255`.
- `BIT`, `BOOL`, `BOOLEAN`

  These are synonyms for `TINYINT(1)`. The `BOOLEAN` synonym was added in MySQL 4.1.0. A value of zero is considered false. Non-zero values are considered true.

  In the future, full boolean type handling will be introduced in accordance with standard SQL.
- `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`.

  Some things you should be aware of with respect to `BIGINT` columns:

  - 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! If you do that, some of the last digits in the result may be wrong because of rounding errors when converting a `BIGINT` value to a `DOUBLE`.
  - MySQL 4.0 can handle `BIGINT` in the following cases:
    - When using integers to store big unsigned values in a BIGINT column.
    - In MIN(col_name) or MAX(col_name), where col_name refers to a BIGINT column.
    - When using operators (+, -, *, and so on) where both operands are integers.
  - You can always store an exact integer value in a `BIGINT` column by storing it using a string. In this case, MySQL performs a string-to-number conversion that involves no intermediate double-precision representation.
  - The `-`, `+`, and `*` operators will use `BIGINT` arithmetic when both operands are integer values! This means that if you multiply two big integers (or results from functions that return integers), you may get unexpected results when the result is larger than `9223372036854775807`.

- `FLOAT(p) [UNSIGNED] [ZEROFILL]`

  A floating-point number. *p* represents the precision. It can be from 0 to 24 for a single-precision floating-point number and from 25 to 53 for a double-precision floating-point number. These types are like the `FLOAT` and `DOUBLE` types described immediately following. `FLOAT(p)` has the same range as the corresponding `FLOAT` and `DOUBLE` types, but the display size and number of decimals are undefined.

  As of MySQL 3.23, this is a true floating-point value. In earlier MySQL versions, `FLOAT(p)` always has two decimals.

  This syntax is provided for ODBC compatibility.

  Using `FLOAT` might give you some unexpected problems because all calculations in MySQL are done with double precision. See Section A.1.7, "Solving Problems with No Matching Rows."

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

  A small (single-precision) floating-point number. Allowable values are `-3.402823466E+38` to `-1.175494351E-38`, `0`, and `1.175494351E-38` to `3.402823466E+38`. If `UNSIGNED` is specified, negative values are disallowed. *M* is the display width and *D* is the number of decimals. `FLOAT` without arguments or `FLOAT(p)` (where *p* is in the range from 0 to 24) stands for a single-precision floating-point number.

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

  A normal-size (double-precision) floating-point number. Allowable values are `-1.7976931348623157E+308` to `-2.2250738585072014E-308`, `0`, and `2.2250738585072014E-308` to `1.7976931348623157E+308`. If `UNSIGNED` is specified, negative values are disallowed. *M* is the display width and *D* is the number of decimals. `DOUBLE` without arguments or `FLOAT(p)` (where *p* is in the range from 25 to 53) stands for a double-precision floating-point number.

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

  `REAL[(M,D)] [UNSIGNED] [ZEROFILL]`

  These are synonyms for `DOUBLE`. Exception: If the server SQL mode includes the `REAL_AS_FLOAT` option, `REAL` is a synonym for `FLOAT` rather than `DOUBLE`.

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

  An unpacked fixed-point number. Behaves like a `CHAR` column; "unpacked" means the number is stored as a string, using one character for each digit of the value. *M* is the total number of digits and *D* is the number of decimals. The decimal point and (for negative numbers) the '-' sign are not counted in *M*, although space for them is reserved. If *D* is 0, values 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*. If `UNSIGNED` is specified, negative values are disallowed.

  If *D* is omitted, the default is 0. If *M* is omitted, the default is 10.

  Prior to MySQL 3.23, the *M* argument must be large enough to include the space needed for the sign and the decimal point.

- `DEC[(`*M*`[,`*D*`])] [UNSIGNED] [ZEROFILL]`

  `NUMERIC[(`*M*`[,`*D*`])] [UNSIGNED] [ZEROFILL]`

  `FIXED[(`*M*`[,`*D*`])] [UNSIGNED] [ZEROFILL]`

  These are synonyms for `DECIMAL`.

  The `FIXED` synonym was added in MySQL 4.1.0 for compatibility with other servers.

## 4.1.2 Overview of Date and Time Types

A summary of the temporal column types follows. For additional information, see Section 4.3, "Date and Time Types." Column storage requirements are given in Section 4.5, "Column Type Storage Requirements."

- `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 partway through the year `2037`.

  A `TIMESTAMP` column is useful for recording the date and time of an `INSERT` or `UPDATE` operation. The first `TIMESTAMP` column in a table is automatically set to the date and time of the most recent operation if you don't assign it a value yourself. You can also set any `TIMESTAMP` column to the current date and time by assigning it a `NULL` value.

  From MySQL 4.1 on, `TIMESTAMP` is returned as a string with the format `'YYYY-MM-DD HH:MM:SS'`. If you want to obtain the value as a number, you should add `+0` to the timestamp column. Different timestamp display widths are not supported.

  In MySQL 4.0 and earlier, `TIMESTAMP` values are displayed 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. The *`M`* argument affects only how a `TIMESTAMP` column is displayed, not storage. Its values always are stored using four bytes each. From MySQL 4.0.12, the `--new` option can be used to make the server behave as in MySQL 4.1.

  Note that `TIMESTAMP(`*M*`)` columns where *`M`* is 8 or 14 are reported to be numbers, whereas other `TIMESTAMP(`*M*`)` columns are reported to be strings. This is just to ensure that you can reliably dump and restore the table with these 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[(2|4)]`

  A year in two-digit or four-digit format. The default is four-digit format. In four-digit format, the allowable values are `1901` to `2155`, and `0000`. In two-digit format, the allowable values are `70` to `69`, representing years from 1970 to 2069. 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 unavailable prior to MySQL 3.22.

## 4.1.3 Overview of String Types

A summary of the string column types follows. For additional information, see Section 4.4, "String Types." Column storage requirements are given in Section 4.5, "Column Type Storage Requirements."

In some cases, MySQL may change a string column to a type different from that given in a `CREATE TABLE` or `ALTER TABLE` statement. See Section 6.2.5.2, "Silent Column Specification Changes."

A change that affects many string column types is that, as of MySQL 4.1, character column definitions can include a `CHARACTER SET` attribute to specify the character set and, optionally, a collation. This applies to `CHAR`, `VARCHAR`, the `TEXT` types, `ENUM`, and `SET`. For example:

```
CREATE TABLE t
(
    c1 CHAR(20) CHARACTER SET utf8,
    c2 CHAR(20) CHARACTER SET latin1 COLLATE latin1_bin
);
```

This table definition creates a column named `c1` that has a character set of `utf8` with the default collation for that character set, and a column named `c2` that has a character set of `latin1` and the binary collation for the character set. The binary collation is not case sensitive.

Character column sorting and comparison are based on the character set assigned to the column. Before MySQL 4.1, sorting and comparison are based on the collation of the server character set. For `CHAR` and `VARCHAR` columns, you can declare the column with the `BINARY` attribute to cause sorting and comparison to be case sensitive using the underlying character code values rather then a lexical ordering.

For more details, see Chapter 3, "Character Set Support."

Also as of 4.1, MySQL interprets length specifications in character column definitions in characters. (Earlier versions interpret them in bytes.)

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

  A fixed-length string that is always right-padded with spaces to the specified length when stored. *`M`* represents the column length. The range of *`M`* is 0 to 255 characters (1 to 255 prior to MySQL 3.23).

  **Note:** Trailing spaces are removed when `CHAR` values are retrieved.

  From MySQL 4.1.0, a `CHAR` column with a length specification greater than 255 is converted to the smallest `TEXT` type that can hold values of the given length. For example, `CHAR(500)` is converted to `TEXT`, and `CHAR(200000)` is converted to `MEDIUMTEXT`. This is a compatibility feature. However, this conversion causes the column to become a variable-length column, and also affects trailing-space removal.

  `CHAR` is shorthand for `CHARACTER`. `NATIONAL CHAR` (or its equivalent short form, `NCHAR`) is the standard SQL way to define that a `CHAR` column should use the default character set. This is the default in MySQL.

  The `BINARY` attribute causes sorting and comparisons to be case sensitive.

  From MySQL 4.1.0 on, the `ASCII` attribute can be specified. It assigns the `latin1` character set to a `CHAR` column.

  From MySQL 4.1.1 on, the `UNICODE` attribute can be specified. It assigns the `ucs2` character set to a `CHAR` column.

  MySQL allows you to create a column of type `CHAR(0)`. This is mainly useful when you have to be compliant with some old applications that depend on the existence of a column but that do not actually use the value. This is also quite nice when you need a column that can take only two values: A `CHAR(0)` column that is not defined as `NOT NULL` occupies only one bit and can take only the values `NULL` and `''` (the empty string).

- `CHAR`

  This is a synonym for `CHAR(1)`.

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

  A variable-length string. *`M`* represents the maximum column length. The range of *`M`* is 0 to 255 characters (1 to 255 prior to MySQL 4.0.2).

  **Note:** Trailing spaces are removed when `VARCHAR` values are stored, which differs from the standard SQL specification.

  From MySQL 4.1.0 on, a `VARCHAR` column with a length specification greater than 255 is converted to the smallest `TEXT` type that can hold values of the given length. For example, `VARCHAR(500)` is converted to `TEXT`, and `VARCHAR(200000)` is converted to `MEDIUMTEXT`. This is a compatibility feature. However, this conversion affects trailing-space removal.

  `VARCHAR` is shorthand for `CHARACTER VARYING`.

  The `BINARY` attribute causes sorting and comparisons to be case sensitive.

- `TINYBLOB`, `TINYTEXT`

  A `BLOB` or `TEXT` column with a maximum length of 255 ($2^8 - 1$) characters.
- `BLOB`, `TEXT`

  A `BLOB` or `TEXT` column with a maximum length of 65,535 ($2^{16} - 1$) characters.
- `MEDIUMBLOB`, `MEDIUMTEXT`

  A `BLOB` or `TEXT` column with a maximum length of 16,777,215 ($2^{24} - 1$) characters.
- `LONGBLOB`, `LONGTEXT`

  A `BLOB` or `TEXT` column with a maximum length of 4,294,967,295 or 4GB ($2^{32} - 1$) characters. Up to MySQL 3.23, the client/server protocol and `MyISAM` tables had a limit of 16MB per communication packet / table row. From MySQL 4.0, the maximum allowed length of `LONGBLOB` or `LONGTEXT` columns depends on the configured maximum packet size in the client/server protocol and available memory.
- `ENUM('`*`value1`*`','`*`value2`*`',...)`

  An enumeration. A string object that can have only one value, chosen from the list of values `'`*`value1`*`'`, `'`*`value2`*`'`, `...`, `NULL` or the special `''` error value. An `ENUM` column can have a maximum of 65,535 distinct values. `ENUM` values are represented internally as integers.
- `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` column can have a maximum of 64 members. `SET` values are represented internally as integers.

## 4.2 Numeric Types

MySQL supports all of the standard SQL numeric data types. These types include the exact numeric data types (`INTEGER`, `SMALLINT`, `DECIMAL`, and `NUMERIC`), as well as the approximate numeric data types (`FLOAT`, `REAL`, and `DOUBLE PRECISION`). The keyword `INT` is a synonym for `INTEGER`, and the keyword `DEC` is a synonym for `DECIMAL`.

As an extension to the SQL standard, MySQL also supports the integer types `TINYINT`, `MEDIUMINT`, and `BIGINT` as listed in the following table.

| Type | Bytes | Minimum Value (Signed) | Maximum Value (Signed) |
|---|---|---|---|
| `TINYINT` | 1 | `-128` | `127` |
| `SMALLINT` | 2 | `-32768` | `32767` |
| `MEDIUMINT` | 3 | `-8388608` | `8388607` |
| `INT` | 4 | `-2147483648` | `2147483647` |
| `BIGINT` | 8 | `-9223372036854775808` | `9223372036854775807` |

Another extension is supported by MySQL for optionally specifying the display width of an integer value in parentheses following the base keyword for the type (for example, `INT(4)`). This optional display width specification is used to left-pad the display of values having a width less than the width specified for the column.

The display width does not constrain the range of values that can be stored in the column, nor the number of digits that will be displayed for values having a width exceeding that specified for the column.

When used in conjunction with the optional extension attribute `ZEROFILL`, the default padding of spaces is replaced with zeros. For example, for a column declared as `INT(5) ZEROFILL`, a value of `4` is retrieved as `00004`. Note that if you store larger values than the display width in an integer column, you may experience problems when MySQL generates temporary tables for some complicated joins, because in these cases MySQL trusts that the data did fit into the original column width.

All integer types can have an optional (non-standard) attribute `UNSIGNED`. Unsigned values can be used when you want to allow only non-negative numbers in a column and you need a bigger upper numeric range for the column.

As of MySQL 4.0.2, floating-point and fixed-point types also can be `UNSIGNED`. As with integer types, this attribute prevents negative values from being stored in the column. However, unlike the integer types, the upper range of column values remains the same.

If you specify `ZEROFILL` for a numeric column, MySQL automatically adds the `UNSIGNED` attribute to the column.

The `DECIMAL` and `NUMERIC` types are implemented as the same type by MySQL. They are used to store values for which it is important to preserve exact precision, for example with monetary data. When declaring a column of one of these types, the precision and scale can be (and usually is) specified; for example:

```
salary DECIMAL(5,2)
```

In this example, `5` is the precision and `2` is the scale. The precision represents the number of significant decimal digits that will be stored for values, and the scale represents the number of digits that will be stored following the decimal point.

MySQL stores `DECIMAL` and `NUMERIC` values as strings, rather than as binary floating-point numbers, in order to preserve the decimal precision of those values. One character is used for each digit of the value, the decimal point (if the scale is greater than 0), and the '-' sign (for negative numbers). If the scale is 0, `DECIMAL` and `NUMERIC` values contain no decimal point or fractional part.

Standard SQL requires that the `salary` column be able to store any value with five digits and two decimals. In this case, therefore, the range of values that can be stored in the `salary` column is from `-999.99` to `999.99`. MySQL varies from this in two ways:

- On the positive end of the range, the column actually can store numbers up to `9999.99`. For positive numbers, MySQL uses the byte reserved for the sign to extend the upper end of the range.
- `DECIMAL` columns in MySQL before 3.23 are stored differently and cannot represent all the values required by standard SQL. This is because for a type of `DECIMAL(`*`M`*`,`*`D`*`)`, the value of *`M`* includes the bytes for the sign and the decimal point. The range of the `salary` column before MySQL 3.23 would be `-9.99` to `99.99`.

In standard SQL, the syntax `DECIMAL(`*`M`*`)` is equivalent to `DECIMAL(`*`M`*`,0)`. Similarly, the syntax `DECIMAL` is equivalent to `DECIMAL(`*`M`*`,0)`, where the implementation is allowed to decide the value of *`M`*. As of MySQL 3.23.6, both of these variant forms of the `DECIMAL` and `NUMERIC` data types are supported. The default value of *`M`* is 10. Before 3.23.6, *`M`* and *`D`* both must be specified explicitly.

The maximum range of `DECIMAL` and `NUMERIC` values is the same as for `DOUBLE`, but the actual range for a given `DECIMAL` or `NUMERIC` column can be constrained by the precision or scale for a given column. When such a column is assigned a value with more digits following the decimal point than are allowed by the specified scale, the value is converted to that scale. (The precise behavior is operating system-specific, but generally the effect is truncation to the allowable number of digits.) When a `DECIMAL` or `NUMERIC` column is assigned a value that exceeds the range implied by the specified (or default) precision and scale, MySQL stores the value representing the corresponding end point of that range.

For floating-point column types, MySQL uses four bytes for single-precision values and eight bytes for double-precision values.

The `FLOAT` type is used to represent approximate numeric data types. The SQL standard allows an optional specification of the precision (but not the range of the exponent) in bits following the keyword `FLOAT` in parentheses. The MySQL implementation also supports this optional precision specification, but the precision value is used only to determine storage size. A precision from 0 to 23 results in four-byte single-precision `FLOAT` column. A precision from 24 to 53 results in eight-byte double-precision `DOUBLE` column.

When the keyword `FLOAT` is used for a column type without a precision specification, MySQL uses four bytes to store the values. MySQL also supports variant syntax with two numbers given in parentheses following the `FLOAT` keyword. The first number represents the display width and the second number specifies the number of digits to be stored and displayed following the decimal point (as with `DECIMAL` and `NUMERIC`). When MySQL is asked to store a number for such a column with more decimal digits following the decimal point than specified for the column, the value is rounded to eliminate the extra digits when the value is stored.

In standard SQL, the REAL and DOUBLE PRECISION types do not accept precision specifications. MySQL supports a variant syntax with two numbers given in parentheses following the type name. The first number represents the display width and the second number specifies the number of digits to be stored and displayed following the decimal point. As an extension to the SQL standard, MySQL recognizes DOUBLE as a synonym for the DOUBLE PRECISION type. In contrast with the standard's requirement that the precision for REAL be smaller than that used for DOUBLE PRECISION, MySQL implements both as eight-byte double-precision floating-point values (unless the server SQL mode includes the REAL_AS_FLOAT option).

For maximum portability, code requiring storage of approximate numeric data values should use FLOAT or DOUBLE PRECISION with no specification of precision or number of decimal points.

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, MySQL clips the value to the lower endpoint of the range and stores -2147483648 instead. Similarly, if you try to insert 9999999999, MySQL clips the value to the upper endpoint of the range and stores 2147483647 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 are 0 and 4294967296.

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

## 4.3 Date and Time Types

The date and time types for representing temporal values are DATETIME, DATE, TIMESTAMP, TIME, and YEAR. Each temporal type has a range of legal values, as well as a "zero" value that is used when you specify an illegal value that MySQL cannot represent. The TIMESTAMP type has special automatic updating behavior, described later on.

MySQL allows you to store certain "not strictly legal" date values, such as '1999-11-31'. The reason for this is that we consider date checking to be the responsibility of the application, not the SQL server. To make date checking faster, MySQL verifies only that the month is in the range from 0 to 12 and that the day is in the range from 0 to 31. These ranges are defined to include zero because MySQL allows you to store dates where the day or month and day are zero in a DATE or DATETIME column. This is extremely useful for applications that need to store a birthdate for which you don't know the exact date. In this case, you simply store the date like '1999-00-00' or '1999-01-00'. If you store dates such as these, you should not expect to get correct results for functions such as DATE_SUB() or DATE_ADD that require complete dates.

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 output format, but it attempts to interpret a variety of formats for input values that you supply (for example, when you specify a value to be assigned to or compared to a date or time type). 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.
- Dates containing two-digit year values are ambiguous because the century is unknown. MySQL interprets two-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`.
- Although MySQL tries to interpret values in several formats, dates always must be given in year-month-day order (for example, `'98-09-04'`), rather than in the month-day-year or day-month-year orders commonly used elsewhere (for example, `'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 (as described at the beginning of this section), 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 following table 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` |
  | `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 Connector/ODBC are converted automatically to `NULL` in Connector/ODBC 2.50.12 and above, because ODBC can't handle such values.

## 4.3.1 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 has varying properties, depending on the MySQL version and the SQL mode the server is running in. These properties are described later in this section.

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 punctuation 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 '971122129015' 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 00000000000000).

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, minute, 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 a 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 completely missing, so the value is not a legal date. However, as of MySQL 3.23, you can explicitly specify a value of zero to represent missing month or day parts. For example, you can use `'990300'` to insert the value `'1999-03-00'`.

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.

- The MySQL server performs only basic checking on the validity of a date: The ranges for year, month, and day are 1000 to 9999, 00 to 12, and 00 to 31, respectively. Any date containing parts not within these ranges is subject to conversion to `'0000-00-00'`. Please note that this still allows you to store invalid dates such as `'2002-04-31'`. To ensure that a date is valid, perform a check in your application.
- Dates containing two-digit year values are ambiguous because the century is unknown. MySQL interprets two-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`.

### 4.3.1.1 `TIMESTAMP` Properties Prior to MySQL 4.1

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 in a table, only the first one is updated automatically.

Automatic updating of the first `TIMESTAMP` column in a table occurs under any of the following conditions:

- You explicitly set the column to `NULL`.
- 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. An `UPDATE` that sets a column to the value it already has does not cause the `TIMESTAMP` column to be updated; if you set a column to its current value, MySQL ignores the update for efficiency.

`TIMESTAMP` columns other than the first can 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 from 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 initializes 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:

```
UPDATE tbl_name
    SET timestamp_col = timestamp_col,
        other_col1 = new_value1,
        other_col2 = new_value2, ...
```

Another way to maintain a column that records row-creation time is 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 partway through 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 following table. The "full" TIMESTAMP format is 14 digits, but TIMESTAMP columns may be created with shorter display sizes:

| Column Type | Display Format |
|---|---|
| TIMESTAMP(14) | YYYYMMDDHHMMSS |
| TIMESTAMP(12) | YYMMDDHHMMSS |
| TIMESTAMP(10) | YYMMDDHHMM |
| TIMESTAMP(8) | YYYYMMDD |
| TIMESTAMP(6) | YYMMDD |
| TIMESTAMP(4) | YYMM |
| TIMESTAMP(2) | 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.

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 is not 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 a function 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.

### 4.3.1.2 `TIMESTAMP` Properties as of MySQL 4.1

From MySQL 4.1.0 on, `TIMESTAMP` properties differ from those of earlier MySQL releases:

- `TIMESTAMP` columns are displayed in the same format as `DATETIME` columns.
- Display widths are not supported in the ways described in the preceding section. In other words, you cannot use `TIMESTAMP(2)`, `TIMESTAMP(4)`, and so on.

In addition, if the MySQL server is running in `MAXDB` mode, `TIMESTAMP` is identical with `DATETIME`. That is, if the server is running in `MAXDB` mode at the time that a table is created, any `TIMESTAMP` columns are created as `DATETIME` columns. As a result, such columns use `DATETIME` display format, have the same range of values, and no automatic updating occurs.

MySQL can be run in `MAXDB` mode as of version 4.1.1. To enable this mode, set the server SQL mode to `MAXDB` at startup using the `--sql-mode=MAXDB` server option or by setting the global `sql_mode` variable at runtime:

```
mysql> SET GLOBAL sql_mode=MAXDB;
```

A client can cause the server to run in `MAXDB` mode for its own connection as follows:

```
mysql> SET SESSION sql_mode=MAXDB;
```

## 4.3.2 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 `'D HH:MM:SS.fraction'` format. You can also use one of the following "relaxed" syntaxes: `'HH:MM:SS.fraction'`, `'HH:MM:SS'`, `'HH:MM'`, `'D HH:MM:SS'`, `'D HH:MM'`, `'D HH'`, or `'SS'`. Here *`D`* represents days and can have a value from 0 to 34. Note that MySQL doesn't yet store the fraction part.
- 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'`. The following alternative formats are also understood: `SS`, `MMSS`, `HHMMSS`, `HHMMSS.fraction`. Note that MySQL doesn't yet store the fraction part.
- 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. Without colons, 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 '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'. TIME values with colons, by contrast, are always treated as time of the day. That is '11:12' will mean '11:12:00', not '00:11:12'.

Values that lie outside the TIME range but are otherwise legal are clipped to the closest 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 because '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.

### 4.3.3 The YEAR Type

The YEAR type is a one-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 from the range for two-digit strings, because 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.

### 4.3.4 Y2K Issues and Date Types

MySQL itself is year 2000 (Y2K) safe (see Section 1.2.5, "Year 2000 Compliance"), but input values presented to MySQL may not be. Any input containing two-digit year values is ambiguous, because the century is unknown. Such values must be interpreted into four-digit form because 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 values mean. If the heuristics used by MySQL do not produce the correct values, you should provide unambiguous input containing four-digit year values.

`ORDER BY` properly sorts `TIMESTAMP` or `YEAR` values that have two-digit years.

Some functions like `MIN()` and `MAX()` will convert a `TIMESTAMP` or `YEAR` to a number. This means that a value with a two-digit year will not work properly with these functions. The fix in this case is to convert the `YEAR` or `TIMESTAMP` to four-digit year format or use something like `MIN(DATE_ADD(timestamp,INTERVAL 0 DAYS))`.

## 4.4 String Types

The string types are `CHAR`, `VARCHAR`, `BLOB`, `TEXT`, `ENUM`, and `SET`. This section describes how these types work and how to use them in your queries.

### 4.4.1 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 from 0 to 255. (Before MySQL 3.23, the length of `CHAR` may be from 1 to 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 from 0 to 255, just as for `CHAR` columns. (Before MySQL 4.0.2, the length of `VARCHAR` may be from 1 to 255.) 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 standard SQL specification.

No lettercase conversion takes place during storage or retrieval.

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

If you need a column for which trailing spaces are not removed, consider using a BLOB or TEXT type. If you want to store binary values such as results from an encryption or compression function that might contain arbitrary byte values, use a BLOB column rather than a CHAR or VARCHAR column to avoid potential problems with trailing space removal that would change data values.

The following table 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.

As of MySQL 4.1, values in CHAR and VARCHAR columns are sorted and compared according to the collation of the character set assigned to the column. Before MySQL 4.1, sorting and comparison are based on the collation of the server character set; you can declare the column with the BINARY attribute to cause sorting and comparison to be case sensitive using the underlying character code values rather then a lexical ordering. BINARY doesn't affect how the column is stored or retrieved.

From MySQL 4.1.0, column type CHAR BYTE is an alias for CHAR BINARY. This is a compatibility feature.

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

From MySQL 4.1.0 on, the ASCII attribute can be specified for CHAR. It assigns the latin1 character set.

From MySQL 4.1.1 on, the UNICODE attribute can be specified for CHAR. It assigns the ucs2 character set.

MySQL may silently change the type of a CHAR or VARCHAR column at table creation time. See Section 6.2.5.2, "Silent Column Specification Changes."

## 4.4.2 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 4.5, "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.

BLOB columns are treated as binary strings, whereas TEXT columns are treated according to their character set. Sorting and comparison for BLOB values is not case sensitive. As of MySQL 4.1, values in TEXT columns are sorted and compared according to the collation of the character set assigned to the column. Before MySQL 4.1, TEXT sorting and comparison are based on the collation of the server character set.

No lettercase conversion takes place during storage or retrieval.

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 ways in which BLOB and TEXT differ from CHAR and VARCHAR are:

- You can have indexes on BLOB and TEXT columns only as of MySQL 3.23.2. Older versions of MySQL did not support indexing these column types.
- For indexes on BLOB and TEXT columns, you must specify an index prefix length. For CHAR and VARCHAR, a prefix length is optional.
- There is no trailing-space removal for BLOB and TEXT columns when values are stored or retrieved. This differs from CHAR columns (trailing spaces are removed when values are retrieved) and from VARCHAR columns (trailing spaces are removed when values are stored).
- BLOB and TEXT columns cannot have DEFAULT values.

From MySQL 4.1.0, LONG and LONG VARCHAR map to the MEDIUMTEXT data type. This is a compatibility feature.

Connector/ODBC defines BLOB values as LONGVARBINARY and TEXT values as LONGVARCHAR.

Because BLOB and TEXT values may be extremely long, you may encounter some constraints in 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,SUBSTRING(comment,20) AS substr
    ->                 ORDER BY substr;
```

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 `--max_sort_length` option when starting the `mysqld` server.

You can group on an expression involving `BLOB` or `TEXT` values by using an alias or by specifying the column position:

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

- The maximum size of a `BLOB` or `TEXT` object is determined by its type, but the largest value you actually can 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 by changing the value of the `max_allowed_packet` variable, but you must do so for both the server and your client program. For example, both `mysql` and `mysqldump` allow you to change the client-side `max_allowed_packet` value.

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.

## 4.4.3 The `ENUM` Type

An `ENUM` is a string object with a value 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. This string can be distinguished from a "normal" empty string by the fact that this string has the numerical value 0. More about this later.
- If an `ENUM` column is declared to allow `NULL`, the `NULL` value is a legal value for the column, and the default value is `NULL`. If an `ENUM` column is declared `NOT NULL`, its 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 here. 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 65,535 elements.

Starting from MySQL 3.23.51, trailing spaces are automatically deleted from `ENUM` member values when the table is created.

Lettercase is irrelevant when you assign values to an `ENUM` column. However, values retrieved from the column later are displayed using the lettercase that was used in the column definition.

If you retrieve an `ENUM` value in a numeric context, the column value's index is returned. For example, you can retrieve numeric values from an `ENUM` column like this:

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

If you store a number into an `ENUM` column, the number is treated as an index, and the value stored is the enumeration member with that index. (However, this will not work with `LOAD DATA`, which treats all input as strings.) It's not advisable to define an `ENUM` column with enumeration values that look like numbers, because this can easily become confusing. For example, the following column has enumeration members with string values of `'0'`, `'1'`, and `'2'`, but numeric index values of `1`, `2`, and `3`:

```
numbers ENUM('0','1','2')
```

`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. To prevent unexpected results, specify the `ENUM` list in alphabetical order. You can also use `GROUP BY CAST(col AS VARCHAR)` or `GROUP BY CONCAT(col)` to make sure that the column is sorted lexically rather than by index number.

If you want to determine all possible values for an `ENUM` column, use `SHOW COLUMNS FROM tbl_name LIKE enum_col` and parse the `ENUM` definition in the second column of the output.

## 4.4.4 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.

Starting from MySQL 3.23.51, trailing spaces are automatically deleted from SET member values when the table is created.

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. For example, you can retrieve numeric values from a SET column like this:

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

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. For a column specified as `SET('a','b','c','d')`, the members have the following decimal and binary 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 does 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. If a column is specified as `SET('a','b','c','d')`, then `'a,d'`, `'d,a'`, and `'d,a,a,d,d'` all will appear as `'a,d'` when retrieved.

If you set a SET column to an unsupported value, the value will be ignored.

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

Normally, you search for a set value using the `FIND_IN_SET()` function or the `LIKE` operator:

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

The first statement finds rows where *set_col* contains the *value* set member. The second is similar, but not the same: It finds rows where *set_col* contains *value* anywhere, even as a substring of another set member.

The following statements also are legal:

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

The first of these statements looks for values containing the first set member. The second looks for an exact match. Be careful with comparisons of the second type. Comparing set values to '*val1*,*val2*' will return different results than comparing values to '*val2*,*val1*'. You should specify the values in the same order they are listed in the column definition.

If you want to determine all possible values for a `SET` column, use `SHOW COLUMNS FROM` *tbl_name* `LIKE` *set_col* and parse the `SET` definition in the second column of the output.

## 4.5 Column Type Storage Requirements

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

The maximum size of a row in a `MyISAM` table is 65,534 bytes. Each `BLOB` and `TEXT` column accounts for only five to nine bytes toward this size.

If a `MyISAM` or `ISAM` table includes any variable-length column types, the record format will also be variable length. 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 6.2.5.2, "Silent Column Specification Changes."

### Storage Requirements for Numeric Types

| Column Type | Storage Required |
|---|---|
| `TINYINT` | 1 byte |
| `SMALLINT` | 2 bytes |
| `MEDIUMINT` | 3 bytes |
| `INT`, `INTEGER` | 4 bytes |
| `BIGINT` | 8 bytes |
| `FLOAT(`*p*`)` | 4 bytes if $0 <= p <= 24$, 8 bytes if $25 <= p <= 53$ |
| `FLOAT` | 4 bytes |
| `DOUBLE [PRECISION]`, `REAL` | 8 bytes |
| `DECIMAL(`*M*,*D*`)`, `NUMERIC(`*M*,*D*`)` | $M$+2 bytes if $D > 0$, $M$+1 bytes if $D = 0$ ($D$+2, if $M < D$) |

## Storage Requirements for Date and Time Types

| Column Type | Storage Required |
|---|---|
| `DATE` | 3 bytes |
| `DATETIME` | 8 bytes |
| `TIMESTAMP` | 4 bytes |
| `TIME` | 3 bytes |
| `YEAR` | 1 byte |

## Storage Requirements for String Types

| Column Type | Storage Required |
|---|---|
| `CHAR(`*`M`*`)` | *M* bytes, $0 <= M <= 255$ |
| `VARCHAR(`*`M`*`)` | *L*+1 bytes, where $L <= M$ and $0 <= M <= 255$ |
| `TINYBLOB`, `TINYTEXT` | *L*+1 bytes, where $L < 2^8$ |
| `BLOB`, `TEXT` | *L*+2 bytes, where $L < 2^{16}$ |
| `MEDIUMBLOB`, `MEDIUMTEXT` | *L*+3 bytes, where $L < 2^{24}$ |
| `LONGBLOB`, `LONGTEXT` | *L*+4 bytes, where $L < 2^{32}$ |
| `ENUM('`*`value1`*`','`*`value2`*`',...)` | 1 or 2 bytes, depending on the number of enumeration values (65,535 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 each, 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. See Section 4.4.2, "The `BLOB` and `TEXT` Types."

The size of an `ENUM` object is determined by the number of different enumeration values. One byte is used for enumerations with up to 255 possible values. Two bytes are used for enumerations with up to 65,535 values. See Section 4.4.3, "The `ENUM` Type."

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. See Section 4.4.4, "The `SET` Type."

## 4.6 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 from `1` to `99999`, `MEDIUMINT UNSIGNED` is the best type. Of the types that represent all the required values, it uses the least amount of storage.

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. (Calculations on `DECIMAL` values may still be done using double-precision operations, however.) 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.

## 4.7 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 following table. These mappings make it easier to import table definitions from other database engines into MySQL:

| Other Vendor Type | MySQL Type |
|---|---|
| `BINARY(`*`M`*`)` | `CHAR(`*`M`*`) BINARY` |
| `CHAR VARYING(`*`M`*`)` | `VARCHAR(`*`M`*`)` |
| `FLOAT4` | `FLOAT` |
| `FLOAT8` | `DOUBLE` |
| `INT1` | `TINYINT` |
| `INT2` | `SMALLINT` |
| `INT3` | `MEDIUMINT` |
| `INT4` | `INT` |
| `INT8` | `BIGINT` |
| `LONG VARBINARY` | `MEDIUMBLOB` |
| `LONG VARCHAR` | `MEDIUMTEXT` |
| `LONG` | `MEDIUMTEXT` (MySQL 4.1.0 on) |
| `MIDDLEINT` | `MEDIUMINT` |
| `VARBINARY(`*`M`*`)` | `VARCHAR(`*`M`*`) BINARY` |

Column type mapping occurs at table creation time, after which the original type specifications are discarded. 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.

# 5

# Functions and Operators

Expressions can be used at several points in SQL statements, such as in the ORDER BY or HAVING clauses of SELECT statements, in the WHERE clause of a SELECT, DELETE, or UPDATE statement, or in SET statements. Expressions can be written using literal values, column values, NULL, functions, and operators. This chapter describes the functions and operators that are allowed for writing expressions in MySQL.

An expression that contains NULL always produces a NULL value unless otherwise indicated in the documentation for a particular function or operator.

**Note:** By default, 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 function arguments are permitted, though.

You can tell the MySQL server to accept spaces after function names by starting it with the `--sql-mode=IGNORE_SPACE` option. Individual client programs can request this behavior by using the CLIENT_IGNORE_SPACE option for `mysql_real_connect()`. In either case, all function names will become reserved words.

For the sake of brevity, most examples in this chapter display the output from the `mysql` program in abbreviated form. Instead of showing examples in this format:

```
mysql> SELECT MOD(29,9);
+-----------+
| mod(29,9) |
+-----------+
|         2 |
+-----------+
1 rows in set (0.00 sec)
```

This format is used instead:

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

## 5.1 Operators

### 5.1.1 Parentheses

- `( ... )`

  Use parentheses to force the order of evaluation in an expression. For example:

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

### 5.1.2 Comparison Operators

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

MySQL performs comparisons using the following rules:

- If one or both arguments are `NULL`, the result of the comparison is `NULL`, except for the `NULL`-safe <=> equality comparison operator.
- If both arguments in a comparison operation are strings, they are compared as strings.
- If both arguments are integers, they are compared as integers.
- Hexadecimal values are treated as binary strings if not compared to a number.
- 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 not case sensitive and use the current character set (ISO-8859-1 Latin1 by default, which also works excellently for English).

The following examples 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
```

Note that when you are comparing a string column with a number, MySQL can't use an index on the column to quickly look up the value. If *str_col* is an indexed string column, the index cannot be used when performing the lookup in the following statement:

```
SELECT * FROM tbl_name WHERE str_col=1;
```

The reason for this is that there are many different strings that may convert to the value `1`: `'1'`, `' 1'`, `'1a'`, ...

- `=`

  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
  ```

- `<=>`

  `NULL`-safe equal. This operator performs an equality comparison like the `=` operator, but returns `1` rather than `NULL` if both operands are `NULL`, and `0` rather than `NULL` if one operand is `NULL`.

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

  `<=>` was added in MySQL 3.23.0.

- `<>`, `!=`

  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;
          -> 0
  ```

- `>=`

  Greater than or equal:

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

- `>`

  Greater than:

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

- `IS NULL`, `IS NOT NULL`

  Tests whether a value is or is not `NULL`.

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

  To be able to work well with ODBC programs, MySQL supports the following extra features when using `IS NULL`:

  - You can find the row that contains the most recent `AUTO_INCREMENT` value by issuing a statement of the following form immediately after generating the value:

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

    This behavior can be disabled by setting `SQL_AUTO_IS_NULL=0`. See Section 6.5.3.1, "`SET` Syntax."

  - For `DATE` and `DATETIME` columns that are declared as `NOT NULL`, you can find the special date `'0000-00-00'` by using a statement like this:

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

    This is needed to get some ODBC applications to work because ODBC doesn't support a `'0000-00-00'` date value.

- `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. Otherwise type conversion takes place according to the rules described at the beginning of this section, but applied to all the three arguments. **Note:** Before MySQL 4.0.5, arguments were converted to the type of *expr* instead.

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

- `expr NOT BETWEEN min AND max`

  This is the same as `NOT (expr BETWEEN min AND max)`.

- `COALESCE(value,...)`

  Returns the first non-`NULL` value in the list.

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

  `COALESCE()` was added in MySQL 3.23.3.

- `GREATEST(value1,value2,...)`

  With two or more arguments, 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'
```

  Before MySQL 3.22.5, you can use `MAX()` instead of `GREATEST()`.

- `expr IN (value,...)`

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

The number of values in the `IN` list is only limited by the `max_allowed_packet` value.

To comply with the SQL standard, from MySQL 4.1 on `IN` returns `NULL` not only if the expression on the left hand side is `NULL`, but also if no match is found in the list and one of the expressions in the list is `NULL`.

From MySQL 4.1 on, `IN()` syntax also is used to write certain types of subqueries. See Section 6.1.8.3, "Subqueries with `ANY`, `IN`, and `SOME`."

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

  This is the same as `NOT (expr IN (value,...))`.

- `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!

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

  Returns `0` if *N* < *N1*, `1` if *N* < *N2* and so on or `-1` if *N* is `NULL`. All arguments are treated as integers. It is required that *N1* < *N2* < *N3* < ... < *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
```

- `LEAST(value1,value2,...)`

  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'
```

Before MySQL 3.22.5, you can use `MIN()` instead of `LEAST()`.

Note that the preceding conversion rules can produce strange results in some borderline cases:

```
mysql> SELECT CAST(LEAST(3600, 9223372036854775808.0) as SIGNED);
        -> -9223372036854775808
```

This happens because MySQL reads `9223372036854775808.0` in an integer context. The integer representation is not good enough to hold the value, so it wraps to a signed integer.

## 5.1.3 Logical Operators

In SQL, all logical operators evaluate to TRUE, FALSE, or `NULL` (UNKNOWN). In MySQL, these are implemented as `1` (TRUE), `0` (FALSE), and `NULL`. Most of this is common to different SQL database servers, although some servers may return any non-zero value for TRUE.

- `NOT`, `!`

  Logical NOT. Evaluates to `1` if the operand is `0`, to `0` if the operand is non-zero, and `NOT NULL` returns `NULL`.

```
mysql> SELECT NOT 10;
        -> 0
mysql> SELECT NOT 0;
        -> 1
mysql> SELECT NOT NULL;
        -> NULL
mysql> SELECT ! (1+1);
        -> 0
mysql> SELECT ! 1+1;
        -> 1
```

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

- AND, &&

  Logical AND. Evaluates to 1 if all operands are non-zero and not NULL, to 0 if one or more operands are 0, otherwise NULL is returned.

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

  Please note that MySQL versions prior to 4.0.5 stop evaluation when a NULL is encountered, rather than continuing the process to check for possible 0 values. This means that in these versions, SELECT (NULL AND 0) returns NULL instead of 0. As of MySQL 4.0.5, the code has been re-engineered so that the result is always as prescribed by the SQL standards while still using the optimization wherever possible.

- OR, ||

  Logical OR. Evaluates to 1 if any operand is non-zero, to NULL if any operand is NULL, otherwise 0 is returned.

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

- XOR

  Logical XOR. Returns NULL if either operand is NULL. For non-NULL operands, evaluates to 1 if an odd number of operands is non-zero, otherwise 0 is returned.

```
mysql> SELECT 1 XOR 1;
        -> 0
mysql> SELECT 1 XOR 0;
        -> 1
mysql> SELECT 1 XOR NULL;
        -> NULL
mysql> SELECT 1 XOR 1 XOR 1;
        -> 1
```

`a XOR b` is mathematically equal to `(a AND (NOT b)) OR ((NOT a) and b)`.

`XOR` was added in MySQL 4.0.2.

### 5.1.4 Case-Sensitivity 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 sensitive even if the column isn't defined as `BINARY` or `BLOB`.

  ```
  mysql> SELECT 'a' = 'A';
          -> 1
  mysql> SELECT BINARY 'a' = 'A';
          -> 0
  ```

  `BINARY` was added in MySQL 3.23.0. As of MySQL 4.0.2, `BINARY str` is a shorthand for `CAST(str AS BINARY)`. See Section 5.7, "Cast Functions."

  Note that in some contexts, if you cast an indexed column to `BINARY`, MySQL will not be able to use the index efficiently.

If you want to compare a `BLOB` value in case-insensitive fashion, you can do so as follows:

- Before MySQL 4.1.1, use the `UPPER()` function to convert the `BLOB` value to uppercase before performing the comparison:

  ```
  SELECT 'A' LIKE UPPER(blob_col) FROM tbl_name;
  ```

  If the comparison value is lowercase, convert the `BLOB` value using `LOWER()` instead.

- For MySQL 4.1.1 and up, `BLOB` columns have a character set of `binary`, which has no concept of lettercase. To perform a case-insensitive comparison, use the `CONVERT()` function to convert the `BLOB` value to a character set that is not case sensitive. The result is a non-binary string, so the `LIKE` operation is not case sensitive:

  ```
  SELECT 'A' LIKE CONVERT(blob_col USING latin1) FROM tbl_name;
  ```

  To use a different character set, substitute its name for `latin1` in the preceding statement.

`CONVERT()` can be used more generally for comparing strings that are represented in different character sets.

## 5.2 Control Flow Functions

- `CASE value WHEN [compare-value] THEN result [WHEN [compare-value] THEN result ...] [ELSE result] END`,

  `CASE WHEN [condition] THEN result [WHEN [condition] THEN result ...] [ELSE result] END`

The first version returns the *result* where *value=compare-value*. The second version returns the result for the first condition that is true. If there was no matching result value, the result after ELSE is returned, or NULL if there is no ELSE part.

```
mysql> SELECT CASE 1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE 'more' END;
        -> 'one'
mysql> SELECT CASE WHEN 1>0 THEN 'true' ELSE 'false' END;
        -> 'true'
mysql> SELECT CASE BINARY 'B' WHEN 'a' THEN 1 WHEN 'b' THEN 2 END;
        -> NULL
```

The type of the return value (INTEGER, DOUBLE, or STRING) is the same as the type of the first returned value (the expression after the first THEN).

CASE was added in MySQL 3.23.3.

- IF(*expr1*,*expr2*,*expr3*)

  If *expr1* is TRUE (*expr1* <> 0 and *expr1* <> NULL) then IF() returns *expr2*, else it returns *expr3*. IF() 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'),'no','yes');
        -> 'no'
```

  If only one of *expr2* or *expr3* is explicitly NULL, the result type of the IF() function is the type of non-NULL expression. (This behavior is new in MySQL 4.0.3.)

  *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 shown, 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.

The default return type of `IF()` (which may matter when it is stored into a temporary table) is calculated in MySQL 3.23 as follows:

| **Expression** | **Return Value** |
|---|---|
| *expr2* or *expr3* returns a string | string |
| *expr2* or *expr3* returns a floating-point value | floating-point |
| *expr2* or *expr3* returns an integer | integer |

If *expr2* and *expr3* are strings, the result is case sensitive if either string is case sensitive (starting from MySQL 3.23.51).

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

  In MySQL 4.0.6 and above, the default result value of `IFNULL(`*expr1*`,`*expr2*`)` is the more "general" of the two expressions, in the order `STRING`, `REAL`, or `INTEGER`. The difference from earlier MySQL versions is mostly notable when you create a table based on expressions or MySQL has to internally store a value from `IFNULL()` in a temporary table.

```
CREATE TABLE tmp SELECT IFNULL(1,'test') AS test;
```

  As of MySQL 4.0.6, the type for the `test` column is `CHAR(4)`, whereas in earlier versions the type would be `BIGINT`.

- `NULLIF(`*expr1*`,`*expr2*`)`

  Returns `NULL` if *expr1* `=` *expr2* is true, else returns *expr1*. This is the same as `CASE WHEN` *expr1* `=` *expr2* `THEN NULL ELSE` *expr1* `END`.

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

  Note that MySQL evaluates *expr1* twice if the arguments are not equal.

  `NULLIF()` was added in MySQL 3.23.15.

## 5.3 String Functions

String-valued functions return `NULL` if the length of the result would be greater than the value of the `max_allowed_packet` system variable.

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

- `ASCII(`*`str`*`)`

  Returns the numeric value of the leftmost character of the string *`str`*. Returns `0` if *`str`* is the empty string. Returns `NULL` if *`str`* is `NULL`. `ASCII()` works for characters with numeric values from `0` to `255`.

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

  See also the `ORD()` function.

- `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'
  ```

- `BIT_LENGTH(`*`str`*`)`

  Returns the length of the string *`str`* in bits.

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

  `BIT_LENGTH()` was added in MySQL 4.0.2.

- `CHAR(`*`N`*`,...)`

  `CHAR()` interprets the arguments as integers and returns a string consisting of the characters given by the 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'
  ```

- `CHAR_LENGTH(`*`str`*`)`

  Returns the length of the string *`str`*, measured in characters. A multi-byte character counts as a single character. This means that for a string containing five two-byte characters, `LENGTH()` returns `10`, whereas `CHAR_LENGTH()` returns `5`.

- `CHARACTER_LENGTH(str)`

  `CHARACTER_LENGTH()` is a synonym for `CHAR_LENGTH()`.

- `COMPRESS(string_to_compress)`

  Compresses a string. This function requires MySQL to have been compiled with a compression library such as `zlib`. Otherwise, the return value is always `NULL`.

  ```
  mysql> SELECT LENGTH(COMPRESS(REPEAT('a',1000)));
          -> 21
  mysql> SELECT LENGTH(COMPRESS(''));
          -> 0
  mysql> SELECT LENGTH(COMPRESS('a'));
          -> 13
  mysql> SELECT LENGTH(COMPRESS(REPEAT('a',16)));
          -> 15
  ```

  The compressed string contents are stored the following way:

  - Empty strings are stored as empty strings.
  - Non-empty strings are stored as a four-byte length of the uncompressed string (low byte first), followed by the compressed string. If the string ends with space, an extra '`.`' character is added to avoid problems with endspace trimming should the result be stored in a `CHAR` or `VARCHAR` column. (Use of `CHAR` or `VARCHAR` to store compressed strings is not recommended. It is better to use a `BLOB` column instead.)

  `COMPRESS()` was added in MySQL 4.1.1.

- `CONCAT(str1,str2,...)`

  Returns the string that results from concatenating the arguments. Returns `NULL` if any argument is `NULL`. May have one or more arguments. A numeric argument is converted to its equivalent string form.

  ```
  mysql> SELECT CONCAT('My', 'S', 'QL');
          -> 'MySQL'
  mysql> SELECT CONCAT('My', NULL, 'QL');
          -> NULL
  mysql> SELECT CONCAT(14.3);
          -> '14.3'
  ```

- `CONCAT_WS(`*`separator`*`,`*`str1`*`,`*`str2`*`,...)`

  `CONCAT_WS()` stands for CONCAT With Separator and is a special form of `CONCAT()`. The first argument is the separator for the rest of the arguments. The separator is added between the strings to be concatenated. The separator can be a string as can the rest of the arguments. If the separator is `NULL`, the result is `NULL`. The function skips any `NULL` values after the separator argument.

```
mysql> SELECT CONCAT_WS(',','First name','Second name','Last Name');
        -> 'First name,Second name,Last Name'
mysql> SELECT CONCAT_WS(',','First name',NULL,'Last Name');
        -> 'First name,Last Name'
```

  Before MySQL 4.0.14, `CONCAT_WS()` skips empty strings as well as `NULL` values.

- `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'
```

- `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'
```

- `EXPORT_SET(`*`bits`*`,`*`on`*`,`*`off`*`,[`*`separator`*`,[`*`number_of_bits`*`]])`

  Returns a string in which for every bit set in *`bits`*, you get an *`on`* string and for every reset bit you get an *`off`* string. Each string is separated by *`separator`* (default ‘,’), and only *`number_of_bits`* (default 64) of *`bits`* is used.

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

- `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 string 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 comma (',') character.

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

- `HEX(`*`N_or_S`*`)`

  If *`N_OR_S`* is a number, 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)`.

  From MySQL 4.0.1 and up, if *`N_OR_S`* is a string, returns a hexadecimal string of *`N_OR_S`* where each character in *`N_OR_S`* is converted to two hexadecimal digits.

```
mysql> SELECT HEX(255);
        -> 'FF'
mysql> SELECT 0x616263;
        -> 'abc'
mysql> SELECT HEX('abc');
        -> 616263
```

- `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'
```

  This function is multi-byte safe.

- 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
  ```

  This function is multi-byte safe. In MySQL 3.23, this function is case sensitive. For 4.0 on, it is case sensitive only if either argument is a binary string.

- LCASE(*str*)

  LCASE() is a synonym for LOWER().

- LEFT(*str*,*len*)

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

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

- LENGTH(*str*)

  Returns the length of the string *str*, measured in bytes. A multi-byte character counts as multiple bytes. This means that for a string containing five two-byte characters, LENGTH() returns 10, whereas CHAR_LENGTH() returns 5.

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

- LOAD_FILE(*file_name*)

  Reads the file and returns the file contents as a string. The file must be located on the server, you must specify the full pathname to the file, and you must have the FILE privilege. The file must be readable by all and be smaller than max_allowed_packet bytes.

  If the file doesn't exist or cannot be read because one of the preceding conditions is not satisfied, the function returns NULL.

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

  Before MySQL 3.23, you must read the file inside your application and create an INSERT statement to update the database with the file contents. If you are using the MySQL++ library, one way to do this can be found in the MySQL++ manual, available at http://dev.mysql.com/doc.

- `LOCATE(substr,str)`

  `LOCATE(substr,str,pos)`

  The first syntax returns the position of the first occurrence of substring *`substr`* in string *`str`*. The second syntax 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');
        -> 4
mysql> SELECT LOCATE('xbar', 'foobar');
        -> 0
mysql> SELECT LOCATE('bar', 'foobarbar',5);
        -> 7
```

  This function is multi-byte safe. In MySQL 3.23, this function is case sensitive. For 4.0 on, it is case sensitive only if either argument is a binary string.

- `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 LOWER('QUADRATICALLY');
        -> 'quadratically'
```

  This function is multi-byte safe.

- `LPAD(str,len,padstr)`

  Returns the string *`str`*, left-padded with the string *`padstr`* to a length of *`len`* characters. If *`str`* is longer than *`len`*, the return value is shortened to *`len`* characters.

```
mysql> SELECT LPAD('hi',4,'??');
        -> '??hi'
mysql> SELECT LPAD('hi',1,'??');
        -> 'h'
```

- `LTRIM(str)`

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

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

  This function is multi-byte safe.

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

  Returns a set value (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, and so on. `NULL` values 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(1 | 4,'hello','nice',NULL,'world');
          -> 'hello'
  mysql> SELECT MAKE_SET(0,'a','b','c');
          -> ''
  ```

- `MID(str,pos,len)`

  `MID(str,pos,len)` is a synonym for `SUBSTRING(str,pos,len)`.

- `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'
  ```

- `OCTET_LENGTH(str)`

  `OCTET_LENGTH()` is a synonym for `LENGTH()`.

- `ORD(str)`

  If the leftmost character of the string `str` is a multi-byte character, returns the code for that character, calculated from the numeric values of its constituent bytes using this formula:

  ```
    (1st byte code * 256)
  + (2nd byte code * 256^2)
  + (3rd byte code * 256^3) ...
  ```

  If the leftmost character is not a multi-byte character, `ORD()` returns the same value as the `ASCII()` function.

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

- `POSITION(substr IN str)`

  `POSITION(substr IN str)` is a synonym for `LOCATE(substr,str)`.

- `QUOTE(str)`

  Quotes a string to produce a result that can be used as a properly escaped data value in an SQL statement. The string is returned surrounded by single quotes and with each instance of single quote (‘'’), backslash (‘\’), ASCII NUL, and Control-Z preceded by a backslash. If the argument is `NULL`, the return value is the word “NULL” without surrounding single quotes. The `QUOTE()` function was added in MySQL 4.0.3.

  ```
  mysql> SELECT QUOTE('Don\'t');
          -> 'Don\'t!'
  mysql> SELECT QUOTE(NULL);
          -> NULL
  ```

- `REPEAT(str,count)`

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

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

- `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'
  ```

  This function is multi-byte safe.

- `REVERSE(str)`

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

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

  This function is multi-byte safe.

- `RIGHT(str,len)`

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

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

  This function is multi-byte safe.

- `RPAD(`*`str`*`,`*`len`*`,`*`padstr`*`)`

  Returns the string *`str`*, right-padded with the string *`padstr`* to a length of *`len`* characters. If *`str`* is longer than *`len`*, the return value is shortened to *`len`* characters.

  ```
  mysql> SELECT RPAD('hi',5,'?');
          -> 'hi???'
  mysql> SELECT RPAD('hi',1,'?');
          -> 'h'
  ```

  This function is multi-byte safe.

- `RTRIM(`*`str`*`)`

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

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

  This function is multi-byte safe.

- `SOUNDEX(`*`str`*`)`

  Returns a soundex string from *`str`*. Two strings that sound almost the same should have identical soundex strings. A standard soundex string is four 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-alphabetic characters are ignored in the given string. All international alphabetic characters outside the A-Z range are treated as vowels.

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

  **Note:** This function implements the original Soundex algorithm, not the more popular enhanced version (also described by D. Knuth). The difference is that the original version discards vowels first and then duplicates, whereas the enhanced version discards duplicates first and then vowels.

- *`expr1`* `SOUNDS LIKE` *`expr2`*

  This is the same as `SOUNDEX(`*`expr1`*`) = SOUNDEX(`*`expr2`*`)`. It is available only in MySQL 4.1 or later.

- `SPACE(`*`N`*`)`

  Returns a string consisting of *`N`* space characters.

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

- SUBSTRING(*str*,*pos*)

  SUBSTRING(*str* FROM *pos*)

  SUBSTRING(*str*,*pos*,*len*)

  SUBSTRING(*str* FROM *pos* FOR *len*)

  The forms without a *len* argument return a substring from string *str* starting at position *pos*. The forms with a *len* argument return a substring *len* characters long from string *str*, starting at position *pos*. The forms that use FROM are standard SQL syntax.

```
mysql> SELECT SUBSTRING('Quadratically',5);
        -> 'ratically'
mysql> SELECT SUBSTRING('foobarbar' FROM 4);
        -> 'barbar'
mysql> SELECT SUBSTRING('Quadratically',5,6);
        -> 'ratica'
```

  This function is multi-byte safe.

- SUBSTRING_INDEX(*str*,*delim*,*count*)

  Returns the substring from string *str* before *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'
```

  This function is multi-byte safe.

- 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 is 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 'barxxyz');
        -> 'barx'
```

  This function is multi-byte safe.

- `UCASE(`*`str`*`)`

  `UCASE()` is a synonym for `UPPER()`.

- `UNCOMPRESS(`*`string_to_uncompress`*`)`

  Uncompresses a string compressed by the `COMPRESS()` function. If the argument is not a compressed value, the result is `NULL`. This function requires MySQL to have been compiled with a compression library such as `zlib`. Otherwise, the return value is always `NULL`.

  ```
  mysql> SELECT UNCOMPRESS(COMPRESS('any string'));
          -> 'any string'
  mysql> SELECT UNCOMPRESS('any string');
          -> NULL
  ```

  `UNCOMPRESS()` was added in MySQL 4.1.1.

- `UNCOMPRESSED_LENGTH(`*`compressed_string`*`)`

  Returns the length of a compressed string before compression.

  ```
  mysql> SELECT UNCOMPRESSED_LENGTH(COMPRESS(REPEAT('a',30)));
          -> 30
  ```

  `UNCOMPRESSED_LENGTH()` was added in MySQL 4.1.1.

- `UNHEX(`*`str`*`)`

  Does the opposite of `HEX(`*`str`*`)`. That is, it interprets each pair of hexadecimal digits in the argument as a number and converts it to the character represented by the number. The resulting characters are returned as a binary string.

  ```
  mysql> SELECT UNHEX('4D7953514C');
          -> 'MySQL'
  mysql> SELECT 0x4D7953514C;
          -> 'MySQL'
  mysql> SELECT UNHEX(HEX('string'));
          -> 'string'
  mysql> SELECT HEX(UNHEX('1267'));
          -> '1267'
  ```

  `UNHEX()` was added in MySQL 4.1.2.

- `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 UPPER('Hej');
          -> 'HEJ'
  ```

  This function is multi-byte safe.

### 5.3.1 String Comparison Functions

MySQL automatically converts numbers to strings as necessary, and vice versa.

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

If you want to convert a number to a string explicitly, use the `CAST()` or `CONCAT()` function:

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

`CAST()` is preferable, but it is unavailable before MySQL 4.0.2.

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 affects only comparisons.

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

- *`expr`* `LIKE` *`pat`* `[ESCAPE '`*`escape-char`*`']`

  Pattern matching using SQL simple regular expression comparison. Returns `1` (TRUE) or `0` (FALSE). If either *`expr`* or *`pat`* is `NULL`, the result is `NULL`.

  With `LIKE` you can use the following two wildcard characters in the pattern:

  | Character | Description |
  |---|---|
  | % | Matches any number of characters, even zero characters |
  | _ | 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.

  | String | Description |
  |---|---|
  | \% | Matches one '%' character |
  | _ | Matches one '_' 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
```

The following two statements illustrate that string comparisons are not case sensitive unless one of the operands is a binary string:

```
mysql> SELECT 'abc' LIKE 'ABC';
        -> 1
mysql> SELECT 'abc' LIKE BINARY 'ABC';
        -> 0
```

In MySQL, `LIKE` is allowed on numeric expressions. (This is an extension to the standard SQL `LIKE`.)

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

**Note:** Because MySQL uses the C escape syntax in strings (for example, '`\n`' to represent newline), 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).

- `expr NOT LIKE pat [ESCAPE 'escape-char']`

  This is the same as `NOT (expr LIKE pat [ESCAPE 'escape-char'])`.

- `expr NOT REGEXP pat`

  `expr NOT RLIKE pat`

  This is the same as `NOT (expr REGEXP pat)`.

- `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. The syntax for regular expressions is discussed in Appendix B, "MySQL Regular Expressions." Returns `1` if `expr` matches `pat`, otherwise returns `0`. If either `expr` or `pat` is `NULL`, the result is `NULL`. `RLIKE` is a synonym for `REGEXP`, provided for `mSQL` compatibility. **Note:** Because MySQL uses the C escape syntax in strings (for example, '`\n`' to represent newline), you must double any '\' that you use in your `REGEXP` strings. As of MySQL 3.23.4, `REGEXP` is not case sensitive for normal (not binary) strings.

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

```
mysql> SELECT 'Monty!' REGEXP '.*';
        -> 1
mysql> SELECT 'new*\n*line' REGEXP 'new\\*.\\*line';
        -> 1
mysql> SELECT 'a' REGEXP 'A', 'a' REGEXP BINARY 'A';
        -> 1  0
mysql> SELECT 'a' REGEXP '^[a-d]';
        -> 1
```

`REGEXP` and `RLIKE` use the current character set (ISO-8859-1 Latin1 by default) when deciding the type of a character. However, these operators are not multi-byte safe.

- `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
```

  As of MySQL 4.0, `STRCMP()` uses the current character set when performing comparisons. This makes the default comparison behavior case insensitive unless one or both of the operands are binary strings. Before MySQL 4.0, `STRCMP()` is case sensitive.

## 5.4 Numeric Functions

### 5.4.1 Arithmetic Operators

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. If one of the arguments is an unsigned integer, and the other argument is also an integer, the result will be an unsigned integer. See Section 5.7, "Cast Functions."

- `+`

  Addition:

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

- `-`

  Subtraction:

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

- `-`

  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 $-2^{63}$!

- `*`

  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!

- `DIV`

  Integer division. Similar to `FLOOR()` but safe with `BIGINT` values.

  ```
  mysql> SELECT 5 DIV 2;
          -> 2
  ```

  `DIV` is new in MySQL 4.1.0.

### 5.4.2 Mathematical Functions

All mathematical functions return `NULL` in case of an error.

- `ABS(X)`

  Returns the absolute value of *X*.

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

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

- `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
```

- `ATAN(`*`Y`*`,`*`X`*`)`

  `ATAN2(`*`Y`*`,`*`X`*`)`

  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 ATAN2(PI(),0);
        -> 1.570796
```

- `CEILING(`*`X`*`)`

  `CEIL(`*`X`*`)`

  Returns the smallest integer value not less than *`X`*.

```
mysql> SELECT CEILING(1.23);
        -> 2
mysql> SELECT CEIL(-1.23);
        -> -1
```

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

  The `CEIL()` alias was added in MySQL 4.0.6.

- `COS(`*`X`*`)`

  Returns the cosine of *`X`*, where *`X`* is given in radians.

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

- `COT(`*`X`*`)`

  Returns the cotangent of *`X`*.

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

- `CRC32(`*`expr`*`)`

  Computes a cyclic redundancy check value and returns a 32-bit unsigned value. The result is `NULL` if the argument is `NULL`. The argument is expected to be a string and will be treated as one if it is not.

```
mysql> SELECT CRC32('MySQL');
        -> 3259397556
```

  `CRC32()` is available as of MySQL 4.1.0.

- `DEGREES(X)`

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

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

- `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
```

- `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`!

- `LN(X)`

  Returns the natural logarithm of *X*.

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

  This function was added in MySQL 4.0.3. It is synonymous with `LOG(X)` in MySQL.

- `LOG(X)`

  `LOG(B,X)`

  If called with one parameter, this function returns the natural logarithm of *X*.

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

  If called with two parameters, this function returns the logarithm of *X* for an arbitrary base *B*.

```
mysql> SELECT LOG(2,65536);
        -> 16.000000
mysql> SELECT LOG(1,100);
        -> NULL
```

  The arbitrary base option was added in MySQL 4.0.3. `LOG(B,X)` is equivalent to `LOG(X)/LOG(B)`.

- `LOG2(X)`

  Returns the base-2 logarithm of *X*.

  ```
  mysql> SELECT LOG2(65536);
          -> 16.000000
  mysql> SELECT LOG2(-100);
          -> NULL
  ```

  `LOG2()` is useful for finding out how many bits a number would require for storage. This function was added in MySQL 4.0.3. In earlier versions, you can use `LOG(X)/LOG(2)` instead.

- `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
  ```

- `MOD(N,M)`

  *N* % *M*

  *N* MOD *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
  mysql> SELECT 29 MOD 9;
          -> 2
  ```

  This function is safe to use with `BIGINT` values. The *N* `MOD` *M* syntax works only as of MySQL 4.1.

- `PI()`

  Returns the value of π. The default number of decimals displayed is five, but MySQL internally uses the full double-precision value for π.

  ```
  mysql> SELECT PI();
          -> 3.141593
  mysql> SELECT PI()+0.000000000000000000;
          -> 3.141592653589793116
  ```

- `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
  ```

- `RADIANS(`*`X`*`)`

  Returns the argument *`X`*, converted from degrees to radians.

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

- `RAND()`

  `RAND(`*`N`*`)`

  Returns a random floating-point value in the range from `0` to `1.0`. If an integer argument *`N`* is specified, it is used as the seed value (producing a repeatable sequence).

  ```
  mysql> SELECT RAND();
          -> 0.9233482386203
  mysql> SELECT RAND(20);
          -> 0.15888261251047
  mysql> SELECT RAND(20);
          -> 0.15888261251047
  mysql> SELECT RAND();
          -> 0.63553050033332
  mysql> SELECT RAND();
          -> 0.70100469486881
  ```

  You can't use a column with `RAND()` values in an `ORDER BY` clause, because `ORDER BY` would evaluate the column multiple times. As of MySQL 3.23, you can retrieve rows in random order like this:

  ```
  mysql> SELECT * FROM tbl_name ORDER BY RAND();
  ```

  `ORDER BY RAND()` combined with `LIMIT` is useful for selecting a random sample of a set of rows:

  ```
  mysql> SELECT * FROM table1, table2 WHERE a=b AND c<d
      -> ORDER BY RAND() LIMIT 1000;
  ```

  Note that `RAND()` in a `WHERE` clause is re-evaluated every time the `WHERE` is executed.

  `RAND()` is not meant to be a perfect random generator, but instead a fast way to generate ad hoc random numbers that will be portable between platforms for the same MySQL version.

- `ROUND(X)`

  `ROUND(X,D)`

  Returns the argument `X`, rounded to the nearest integer. With two arguments, returns `X` rounded to `D` decimals. If `D` is negative, the integer part of the number is zeroed out.

```
mysql> SELECT ROUND(-1.23);
        -> -1
mysql> SELECT ROUND(-1.58);
        -> -2
mysql> SELECT ROUND(1.58);
        -> 2
mysql> SELECT ROUND(1.298, 1);
        -> 1.3
mysql> SELECT ROUND(1.298, 0);
        -> 1
mysql> SELECT ROUND(23.298, -1);
        -> 20
```

  Note that the behavior of `ROUND()` when the argument is halfway between two integers depends on the C library implementation. Different implementations round to the nearest even number, always up, always down, or always toward zero. If you need one kind of rounding, you should use a well-defined function such as `TRUNCATE()` or `FLOOR()` instead.

- `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
```

- `SIN(X)`

  Returns the sine of `X`, where `X` is given in radians.

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

- `SQRT(X)`

  Returns the non-negative square root of `X`.

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

- `TAN(X)`

  Returns the tangent of `X`, where `X` is given in radians.

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

- `TRUNCATE(X,D)`

  Returns the number `X`, truncated to `D` decimals. If `D` is `0`, the result will have no decimal point or fractional part. If `D` is negative, the integer part of the number is zeroed out.

  ```
  mysql> SELECT TRUNCATE(1.223,1);
          -> 1.2
  mysql> SELECT TRUNCATE(1.999,1);
          -> 1.9
  mysql> SELECT TRUNCATE(1.999,0);
          -> 1
  mysql> SELECT TRUNCATE(-1.999,1);
          -> -1.9
  mysql> SELECT TRUNCATE(122,-2);
         -> 100
  ```

  Starting from MySQL 3.23.51, all numbers are rounded toward zero.

  Note that decimal numbers are normally not stored as exact numbers in computers, but as double-precision values, so you may be surprised by the following result:

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

  This happens because `10.28` is actually stored as something like `10.2799999999999999`.

## 5.5 Date and Time Functions

This section describes the functions that can be used to manipulate temporal values. See Section 4.3, "Date and Time Types," for a description of the range of values each date and time type has and the valid formats in which values may be specified.

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

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

Note that the query also will select records with dates that lie in the future.

Functions that expect date values usually will accept datetime values and ignore the time part. Functions that expect time values usually will accept datetime values and ignore the date part.

Functions that return the current date or time each are evaluated only once per query at the start of query execution. This means that multiple references to a function such as `NOW()` within a single query will always produce the same result. This principle also applies to `CURDATE()`, `CURTIME()`, `UTC_DATE()`, `UTC_TIME()`, `UTC_TIMESTAMP()`, and to any of their synonyms.

The return value ranges in the following function descriptions apply for complete dates. If a date is a "zero" value or an incomplete date such as `'2001-11-00'`, functions that extract a part of a date may return `0`. For example, `DAYOFMONTH('2001-11-00')` returns `0`.

- `ADDDATE(`*`date`*`,INTERVAL `*`expr type`*`)`

  `ADDDATE(`*`expr`*`,`*`days`*`)`

  When invoked with the `INTERVAL` form of the second argument, `ADDDATE()` is a synonym for `DATE_ADD()`. The related function `SUBDATE()` is a synonym for `DATE_SUB()`. For information on the `INTERVAL` argument, see the discussion for `DATE_ADD()`.

  ```
  mysql> SELECT DATE_ADD('1998-01-02', INTERVAL 31 DAY);
          -> '1998-02-02'
  mysql> SELECT ADDDATE('1998-01-02', INTERVAL 31 DAY);
          -> '1998-02-02'
  ```

  As of MySQL 4.1.1, the second syntax is allowed, where *`expr`* is a date or datetime expression and *`days`* is the number of days to be added to *`expr`*.

  ```
  mysql> SELECT ADDDATE('1998-01-02', 31);
          -> '1998-02-02'
  ```

- `ADDTIME(`*`expr`*`,`*`expr2`*`)`

  `ADDTIME()` adds *`expr2`* to *`expr`* and returns the result. *`expr`* is a date or datetime expression, and *`expr2`* is a time expression.

  ```
  mysql> SELECT ADDTIME('1997-12-31 23:59:59.999999',
      ->                '1 1:1:1.000002');
          -> '1998-01-02 01:01:01.000001'
  mysql> SELECT ADDTIME('01:00:00.999999', '02:00:00.999998');
          -> '03:00:01.999997'
  ```

  `ADDTIME()` was added in MySQL 4.1.1.

- `CURDATE()`

  Returns the current 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
  ```

- `CURRENT_DATE`, `CURRENT_DATE()`

  `CURRENT_DATE` and `CURRENT_DATE()` are synonyms for `CURDATE()`.

- `CURTIME()`

  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
  ```

- `CURRENT_TIME`, `CURRENT_TIME()`

  `CURRENT_TIME` and `CURRENT_TIME()` are synonyms for `CURTIME()`.

- `CURRENT_TIMESTAMP`, `CURRENT_TIMESTAMP()`

  `CURRENT_TIMESTAMP` and `CURRENT_TIMESTAMP()` are synonyms for `NOW()`.

- `DATE(`*`expr`*`)`

  Extracts the date part of the date or datetime expression *`expr`*.

  ```
  mysql> SELECT DATE('2003-12-31 01:02:03');
          -> '2003-12-31'
  ```

  `DATE()` is available as of MySQL 4.1.1.

- `DATEDIFF(`*`expr`*`,`*`expr2`*`)`

  `DATEDIFF()` returns the number of days between the start date *`expr`* and the end date *`expr2`*. *`expr`* and *`expr2`* are date or date-and-time expressions. Only the date parts of the values are used in the calculation.

  ```
  mysql> SELECT DATEDIFF('1997-12-31 23:59:59','1997-12-30');
          -> 1
  mysql> SELECT DATEDIFF('1997-11-30 23:59:59','1997-12-31');
          -> -31
  ```

  `DATEDIFF()` was added in MySQL 4.1.1.

- `DATE_ADD(`*`date`*`,INTERVAL `*`expr type`*`)`

  `DATE_SUB(`*`date`*`,INTERVAL `*`expr type`*`)`

  These functions perform date arithmetic. *`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 `INTERVAL` keyword and the *`type`* specifier are not case sensitive.

The following table shows how the *type* and *expr* arguments are related:

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

The *type* values DAY_MICROSECOND, HOUR_MICROSECOND, MINUTE_MICROSECOND, SECOND_MICROSECOND, and MICROSECOND are allowed as of MySQL 4.1.1. The values QUARTER and WEEK are allowed as of MySQL 5.0.0.

MySQL allows any punctuation delimiter in the *expr* format. Those 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.

As of MySQL 3.23, INTERVAL *expr type* is allowed on either side of the + operator if the expression on the other side is a date or datetime value. For the - operator, INTERVAL *expr type* is allowed only on the right side, because it makes no sense to subtract a date or datetime value from an interval. (See examples below.)

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

```
mysql> SELECT DATE_ADD('1997-12-31 23:59:59',
    ->                 INTERVAL 1 SECOND);
        -> '1998-01-01 00:00:00'
mysql> SELECT DATE_ADD('1997-12-31 23:59:59',
    ->                 INTERVAL 1 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'
mysql> SELECT DATE_ADD('1992-12-31 23:59:59.000002',
    ->            INTERVAL '1.999999' SECOND_MICROSECOND);
        -> '1993-01-01 00:00:01.000001'
```

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 that 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 add to or subtract from a date value something that contains a time part, the result is automatically converted to a datetime value:

```
mysql> SELECT DATE_ADD('1999-01-01', INTERVAL 1 DAY);
        -> '1999-01-02'
mysql> SELECT DATE_ADD('1999-01-01', INTERVAL 1 HOUR);
        -> '1999-01-01 01:00:00'
```

If you use really malformed 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'
```

- `DATE_FORMAT(`*`date`*`,`*`format`*`)`

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

| Specifier | Description |
|---|---|
| `%a` | Abbreviated weekday name (`Sun`..`Sat`) |
| `%b` | Abbreviated month name (`Jan`..`Dec`) |
| `%c` | Month, numeric (0..12) |
| `%D` | Day of the month with English suffix (`0th`, `1st`, `2nd`, `3rd`, ...) |
| `%d` | Day of the month, numeric (00..31) |
| `%e` | Day of the month, numeric (0..31) |
| `%f` | Microseconds (000000..999999) |
| `%H` | Hour (00..23) |
| `%h` | Hour (01..12) |
| `%I` | Hour (01..12) |
| `%i` | Minutes, numeric (00..59) |
| `%j` | Day of year (001..366) |
| `%k` | Hour (0..23) |
| `%l` | Hour (1..12) |
| `%M` | Month name (`January`..`December`) |
| `%m` | Month, numeric (00..12) |
| `%p` | `AM` or `PM` |
| `%r` | Time, 12-hour (`hh:mm:ss` followed by `AM` or `PM`) |
| `%S` | Seconds (00..59) |
| `%s` | Seconds (00..59) |
| `%T` | Time, 24-hour (`hh:mm:ss`) |
| `%U` | Week (00..53), where Sunday is the first day of the week |
| `%u` | Week (00..53), where Monday is the first day of the week |
| `%V` | Week (01..53), where Sunday is the first day of the week; used with `%X` |
| `%v` | Week (01..53), where Monday is the first day of the week; used with `%x` |
| `%W` | Weekday name (`Sunday`..`Saturday`) |
| `%w` | Day of the week (0=Sunday..6=Saturday) |
| `%X` | Year for the week where Sunday is the first day of the week, numeric, four digits; used with `%V` |
| `%x` | Year for the week, where Monday is the first day of the week, numeric, four digits; used with `%v` |
| `%Y` | Year, numeric, four digits |
| `%y` | Year, numeric, two digits |
| `%%` | A literal '%'. |

All other characters are copied to the result without interpretation.

The `%v`, `%V`, `%x`, and `%X` format specifiers are available as of MySQL 3.23.8. `%f` is available as of MySQL 4.1.1.

As of MySQL 3.23, the '`%`' character is required before format specifier characters. In earlier versions of MySQL, '`%`' was optional.

The reason the ranges for the month and day specifiers begin with zero is that MySQL allows incomplete dates such as `'2004-00-00'` to be stored as of MySQL 3.23.

```
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00', '%W %M %Y');
        -> 'Saturday October 1997'
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00', '%H:%i:%s');
        -> '22:23:00'
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00',
                          '%D %y %a %d %m %b %j');
        -> '4th 97 Sat 04 10 Oct 277'
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00',
                          '%H %k %I %r %T %S %w');
        -> '22 22 10 10:23:00 PM 22:23:00 00 6'
mysql> SELECT DATE_FORMAT('1999-01-01', '%X %V');
        -> '1998 52'
```

- `DAY(`*`date`*`)`

  `DAY()` is a synonym for `DAYOFMONTH()`. It is available as of MySQL 4.1.1.

- `DAYNAME(`*`date`*`)`

  Returns the name of the weekday for *`date`*.

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

- `DAYOFMONTH(`*`date`*`)`

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

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

- `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
```

- `DAYOFYEAR(`*`date`*`)`

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

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

- `EXTRACT(`*`type`*` FROM `*`date`*`)`

  The `EXTRACT()` function uses the same kinds of interval type specifiers as `DATE_ADD()` or `DATE_SUB()`, but extracts parts from the date rather than performing date arithmetic. See the description of `DATE_ADD()` for information about *`type`* values.

  ```
  mysql> SELECT EXTRACT(YEAR FROM '1999-07-02');
         -> 1999
  mysql> SELECT EXTRACT(YEAR_MONTH FROM '1999-07-02 01:02:03');
         -> 199907
  mysql> SELECT EXTRACT(DAY_MINUTE FROM '1999-07-02 01:02:03');
         -> 20102
  mysql> SELECT EXTRACT(MICROSECOND
      ->                FROM '2003-01-02 10:30:00.00123');
          -> 123
  ```

  `EXTRACT()` was added in MySQL 3.23.0.

- `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), because it does not take into account the days that were lost when the calendar was changed.

- `FROM_UNIXTIME(`*`unix_timestamp`*`)`

  `FROM_UNIXTIME(`*`unix_timestamp`*`,`*`format`*`)`

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

  If *`format`* is given, the result is 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');
          -> '2003 6th August 06:22:58 2003'
  ```

- `GET_FORMAT(DATE|TIME|TIMESTAMP, 'EUR'|'USA'|'JIS'|'ISO'|'INTERNAL')`

  Returns a format string. This function is useful in combination with the `DATE_FORMAT()` and the `STR_TO_DATE()` functions.

  The three possible values for the first argument and the five possible values for the second argument result in 15 possible format strings (for the specifiers used, see the table in the `DATE_FORMAT()` function description).

| Function Call | Result |
|---|---|
| `GET_FORMAT(DATE,'USA')` | `'%m.%d.%Y'` |
| `GET_FORMAT(DATE,'JIS')` | `'%Y-%m-%d'` |
| `GET_FORMAT(DATE,'ISO')` | `'%Y-%m-%d'` |
| `GET_FORMAT(DATE,'EUR')` | `'%d.%m.%Y'` |
| `GET_FORMAT(DATE,'INTERNAL')` | `'%Y%m%d'` |
| `GET_FORMAT(TIMESTAMP,'USA')` | `'%Y-%m-%d-%H.%i.%s'` |
| `GET_FORMAT(TIMESTAMP,'JIS')` | `'%Y-%m-%d %H:%i:%s'` |
| `GET_FORMAT(TIMESTAMP,'ISO')` | `'%Y-%m-%d %H:%i:%s'` |
| `GET_FORMAT(TIMESTAMP,'EUR')` | `'%Y-%m-%d-%H.%i.%s'` |
| `GET_FORMAT(TIMESTAMP,'INTERNAL')` | `'%Y%m%d%H%i%s'` |
| `GET_FORMAT(TIME,'USA')` | `'%h:%i:%s %p'` |
| `GET_FORMAT(TIME,'JIS')` | `'%H:%i:%s'` |
| `GET_FORMAT(TIME,'ISO')` | `'%H:%i:%s'` |
| `GET_FORMAT(TIME,'EUR')` | `'%H.%i.%S'` |
| `GET_FORMAT(TIME,'INTERNAL')` | `'%H%i%s'` |

  ISO format is ISO 9075, not ISO 8601.

```
mysql> SELECT DATE_FORMAT('2003-10-03',GET_FORMAT(DATE,'EUR'));
        -> '03.10.2003'
mysql> SELECT STR_TO_DATE('10.31.2003',GET_FORMAT(DATE,'USA'));
        -> 2003-10-31
```

  `GET_FORMAT()` is available as of MySQL 4.1.1. See Section 6.5.3.1, "`SET` Syntax."

- `HOUR(`*`time`*`)`

  Returns the hour for *`time`*. The range of the return value will be `0` to `23` for time-of-day values.

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

  However, the range of `TIME` values actually is much larger, so `HOUR` can return values greater than `23`.

```
mysql> SELECT HOUR('272:59:59');
        -> 272
```

- `LAST_DAY(`*`date`*`)`

  Takes a date or datetime value and returns the corresponding value for the last day of the month. Returns `NULL` if the argument is invalid.

  ```
  mysql> SELECT LAST_DAY('2003-02-05');
          -> '2003-02-28'
  mysql> SELECT LAST_DAY('2004-02-05');
          -> '2004-02-29'
  mysql> SELECT LAST_DAY('2004-01-01 01:01:01');
          -> '2004-01-31'
  mysql> SELECT LAST_DAY('2003-03-32');
          -> NULL
  ```

  `LAST_DAY()` is available as of MySQL 4.1.1.

- `LOCALTIME`, `LOCALTIME()`

  `LOCALTIME` and `LOCALTIME()` are synonyms for `NOW()`. They were added in MySQL 4.0.6.

- `LOCALTIMESTAMP`, `LOCALTIMESTAMP()`

  `LOCALTIMESTAMP` and `LOCALTIMESTAMP()` are synonyms for `NOW()`. They were added in MySQL 4.0.6.

- `MAKEDATE(`*`year`*`,`*`dayofyear`*`)`

  Returns a date, given year and day-of-year values. *`dayofyear`* must be greater than 0 or the result will be `NULL`.

  ```
  mysql> SELECT MAKEDATE(2001,31), MAKEDATE(2001,32);
          -> '2001-01-31', '2001-02-01'
  mysql> SELECT MAKEDATE(2001,365), MAKEDATE(2004,365);
          -> '2001-12-31', '2004-12-30'
  mysql> SELECT MAKEDATE(2001,0);
          -> NULL
  ```

  `MAKEDATE()` is available as of MySQL 4.1.1.

- `MAKETIME(`*`hour`*`,`*`minute`*`,`*`second`*`)`

  Returns a time value calculated from the *`hour`*, *`minute`*, and *`second`* arguments.

  ```
  mysql> SELECT MAKETIME(12,15,30);
          -> '12:15:30'
  ```

  `MAKETIME()` is available as of MySQL 4.1.1.

- `MICROSECOND(expr)`

  Returns the microseconds from the time or datetime expression *`expr`* as a number in the range from `0` to `999999`.

  ```
  mysql> SELECT MICROSECOND('12:00:00.123456');
          -> 123456
  mysql> SELECT MICROSECOND('1997-12-31 23:59:59.000010');
          -> 10
  ```

  `MICROSECOND()` is available as of MySQL 4.1.1.

- `MINUTE(time)`

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

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

- `MONTH(date)`

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

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

- `MONTHNAME(date)`

  Returns the full name of the month for *`date`*.

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

- `NOW()`

  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
  ```

- `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
  ```

- `QUARTER(`*`date`*`)`

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

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

- `SECOND(`*`time`*`)`

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

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

- `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
```

- `STR_TO_DATE(`*`str`*`,`*`format`*`)`

  This is the reverse function of the `DATE_FORMAT()` function. It takes a string *`str`* and a format string *`format`*, and returns a `DATETIME` value.

  The date, time, or datetime values contained in *`str`* should be given in the format indicated by *`format`*. For the specifiers that can be used in *`format`*, see the table in the `DATE_FORMAT()` function description. All other characters are just taken verbatim, thus not being interpreted. If *`str`* contains an illegal date, time, or datetime value, `STR_TO_DATE()` returns `NULL`.

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

  `STR_TO_DATE()` is available as of MySQL 4.1.1.

- `SUBDATE(`*`date`*`,INTERVAL `*`expr type`*`)`

  `SUBDATE(`*`expr`*`,`*`days`*`)`

  When invoked with the `INTERVAL` form of the second argument, `SUBDATE()` is a synonym for `DATE_SUB()`. For information on the `INTERVAL` argument, see the discussion for `DATE_ADD()`.

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

  As of MySQL 4.1.1, the second syntax is allowed, where *`expr`* is a date or datetime expression and *`days`* is the number of days to be subtracted from *`expr`*.

  ```
  mysql> SELECT SUBDATE('1998-01-02 12:00:00', 31);
          -> '1997-12-02 12:00:00'
  ```

- `SUBTIME(`*`expr`*`,`*`expr2`*`)`

  `SUBTIME()` subtracts *`expr2`* from *`expr`* and returns the result. *`expr`* is a date or datetime expression, and *`expr2`* is a time expression.

  ```
  mysql> SELECT SUBTIME('1997-12-31 23:59:59.999999',
      ->                '1 1:1:1.000002');
          -> '1997-12-30 22:58:58.999997'
  mysql> SELECT SUBTIME('01:00:00.999999', '02:00:00.999998');
          -> '-00:59:59.999999'
  ```

  `SUBTIME()` was added in MySQL 4.1.1.

- `SYSDATE()`

  `SYSDATE()` is a synonym for `NOW()`.

- `TIME(`*`expr`*`)`

  Extracts the time part of the time or datetime expression *`expr`*.

  ```
  mysql> SELECT TIME('2003-12-31 01:02:03');
          -> '01:02:03'
  mysql> SELECT TIME('2003-12-31 01:02:03.000123');
          -> '01:02:03.000123'
  ```

  `TIME()` is available as of MySQL 4.1.1.

- `TIMEDIFF(`*`expr`*`,`*`expr2`*`)`

  `TIMEDIFF()` returns the time between the start time *`expr`* and the end time *`expr2`*. *`expr`* and *`expr2`* are time or date-and-time expressions, but both must be of the same type.

  ```
  mysql> SELECT TIMEDIFF('2000:01:01 00:00:00',
      ->                 '2000:01:01 00:00:00.000001');
          -> '-00:00:00.000001'
  mysql> SELECT TIMEDIFF('1997-12-31 23:59:59.000001',
      ->                 '1997-12-30 01:01:01.000002');
          -> '46:58:57.999999'
  ```

  `TIMEDIFF()` was added in MySQL 4.1.1.

- `TIMESTAMP(`*`expr`*`)`

  `TIMESTAMP(`*`expr`*`,`*`expr2`*`)`

  With one argument, returns the date or datetime expression *`expr`* as a datetime value. With two arguments, adds the time expression *`expr2`* to the date or datetime expression *`expr`* and returns a datetime value.

  ```
  mysql> SELECT TIMESTAMP('2003-12-31');
          -> '2003-12-31 00:00:00'
  mysql> SELECT TIMESTAMP('2003-12-31 12:00:00','12:00:00');
          -> '2004-01-01 00:00:00'
  ```

  `TIMESTAMP()` is available as of MySQL 4.1.1.

- `TIMESTAMPADD(`*`interval`*`,`*`int_expr`*`,`*`datetime_expr`*`)`

  Adds the integer expression *`int_expr`* to the date or datetime expression *`datetime_expr`*. The unit for *`int_expr`* is given by the *`interval`* argument, which should be one of the following values: `FRAC_SECOND`, `SECOND`, `MINUTE`, `HOUR`, `DAY`, `WEEK`, `MONTH`, `QUARTER`, or `YEAR`.

  The *`interval`* value may be specified using one of keywords as shown, or with a prefix of `SQL_TSI_`. For example, `DAY` or `SQL_TSI_DAY` both are legal.

  ```
  mysql> SELECT TIMESTAMPADD(MINUTE,1,'2003-01-02');
          -> '2003-01-02 00:01:00'
  mysql> SELECT TIMESTAMPADD(WEEK,1,'2003-01-02');
          -> '2003-01-09'
  ```

  `TIMESTAMPADD()` is available as of MySQL 5.0.0.

- `TIMESTAMPDIFF(`*`interval`*`,`*`datetime_expr1`*`,`*`datetime_expr2`*`)`

  Returns the integer difference between the date or datetime expressions *`datetime_expr1`* and *`datetime_expr2`*. The unit for the result is given by the *`interval`* argument. The legal values for *`interval`* are the same as those listed in the description of the `TIMESTAMPADD()` function.

  ```
  mysql> SELECT TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01');
          -> 3
  mysql> SELECT TIMESTAMPDIFF(YEAR,'2002-05-01','2001-01-01');
          -> -1
  ```

  `TIMESTAMPDIFF()` is available as of MySQL 5.0.0.

- `TIME_FORMAT(`*`time`*`,`*`format`*`)`

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

  If the *`time`* value contains an hour part that is greater than `23`, the `%H` and `%k` hour format specifiers produce a value larger than the usual range of `0..23`. The other hour format specifiers produce the hour value modulo 12.

  ```
  mysql> SELECT TIME_FORMAT('100:00:00', '%H %k %h %I %l');
          -> '100 100 04 04 4'
  ```

- `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
  ```

- `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), because it does not take into account the days that were lost when the calendar was changed.

Remember that MySQL converts two-digit year values in dates to four-digit form using the rules in Section 4.3, "Date and Time Types." For example, `'1997-10-07'` and `'97-10-07'` are seen as identical dates:

```
mysql> SELECT TO_DAYS('1997-10-07'), TO_DAYS('97-10-07');
        -> 729669, 729669
```

For other dates before 1582, results from this function are undefined.

- `UNIX_TIMESTAMP()`

  `UNIX_TIMESTAMP(`*`date`*`)`

  If called with no argument, returns a Unix timestamp (seconds since `'1970-01-01 00:00:00'` GMT) as an unsigned integer. 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 returns the internal timestamp value directly, with no implicit "string-to-Unix-timestamp" conversion. If you pass an out-of-range date to `UNIX_TIMESTAMP()`, it returns `0`, but please note that only basic range checking is performed (year from `1970` to `2037`, month from `01` to `12`, day from `01` from `31`).

  If you want to subtract `UNIX_TIMESTAMP()` columns, you might want to cast the result to signed integers. See Section 5.7, "Cast Functions."

- `UTC_DATE`, `UTC_DATE()`

  Returns the current UTC 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 UTC_DATE(), UTC_DATE() + 0;
        -> '2003-08-14', 20030814
```

  `UTC_DATE()` is available as of MySQL 4.1.1.

- `UTC_TIME`, `UTC_TIME()`

  Returns the current UTC 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 UTC_TIME(), UTC_TIME() + 0;
        -> '18:07:53', 180753
```

  `UTC_TIME()` is available as of MySQL 4.1.1.

- `UTC_TIMESTAMP`, `UTC_TIMESTAMP()`

  Returns the current UTC 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 UTC_TIMESTAMP(), UTC_TIMESTAMP() + 0;
        -> '2003-08-14 18:08:04', 20030814180804
```

  `UTC_TIMESTAMP()` is available as of MySQL 4.1.1.

- `WEEK(`*`date`*`[,`*`mode`*`])`

  The function returns the week number for *`date`*. The two-argument form of `WEEK()` allows you to specify whether the week starts on Sunday or Monday and whether the return value should be in the range from `0` to `53` or from `1` to `52`. If the *`mode`* argument is omitted, the value of the `default_week_format` system variable is used (or `0` before MySQL 4.0.14).

  The following table describes how the *`mode`* argument works:

| Value | Meaning |
|---|---|
| 0 | Week starts on Sunday; return value range is `0` to `53`; week 1 is the first week that starts in this year |
| 1 | Week starts on Monday; return value range is `0` to `53`; week 1 is the first week that has more than three days in this year |
| 2 | Week starts on Sunday; return value range is `1` to `53`; week 1 is the first week that starts in this year |
| 3 | Week starts on Monday; return value range is `1` to `53`; week 1 is the first week that has more than three days in this year |
| 4 | Week starts on Sunday; return value range is `0` to `53`; week 1 is the first week that has more than three days in this year |
| 5 | Week starts on Monday; return value range is `0` to `53`; week 1 is the first week that starts in this year |
| 6 | Week starts on Sunday; return value range is `1` to `53`; week 1 is the first week that has more than three days in this year |
| 7 | Week starts on Monday; return value range is `1` to `53`; week 1 is the first week that starts in this year |

  The *`mode`* value of `3` can be used as of MySQL 4.0.5. Values of `4` and above can be used as of MySQL 4.0.17.

```
mysql> SELECT WEEK('1998-02-20');
        -> 7
mysql> SELECT WEEK('1998-02-20',0);
        -> 7
mysql> SELECT WEEK('1998-02-20',1);
        -> 8
mysql> SELECT WEEK('1998-12-31',1);
        -> 53
```

**Note:** In MySQL 4.0, `WEEK(*date*,0)` was changed to match the calendar in the USA. Before that, `WEEK()` was calculated incorrectly for dates in the USA. (In effect, `WEEK(*date*)` and `WEEK(*date*,0)` were incorrect for all cases.)

Note that if a date falls in the last week of the previous year, MySQL returns `0` if you don't use `2`, `3`, `6`, or `7` as the optional *`mode`* argument:

```
mysql> SELECT YEAR('2000-01-01'), WEEK('2000-01-01',0);
        -> 2000, 0
```

One might argue that MySQL should return `52` for the `WEEK()` function, because the given date actually occurs in the 52nd week of 1999. We decided to return `0` instead because we want the function to return "the week number in the given year." This makes use of the `WEEK()` function reliable when combined with other functions that extract a date part from a date.

If you would prefer the result to be evaluated with respect to the year that contains the first day of the week for the given date, you should use `2`, `3`, `6`, or `7` as the optional *`mode`* argument.

```
mysql> SELECT WEEK('2000-01-01',2);
        -> 52
```

Alternatively, use the `YEARWEEK()` function:

```
mysql> SELECT YEARWEEK('2000-01-01');
        -> 199952
mysql> SELECT MID(YEARWEEK('2000-01-01'),5,2);
        -> '52'
```

- `WEEKDAY(*date*)`

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

```
mysql> SELECT WEEKDAY('1998-02-03 22:23:00');
        -> 1
mysql> SELECT WEEKDAY('1997-11-05');
        -> 2
```

- `WEEKOFYEAR(*date*)`

  Returns the calendar week of the date as a number in the range from `1` to `53`.

```
mysql> SELECT WEEKOFYEAR('1998-02-20');
        -> 8
```

  `WEEKOFYEAR()` is available as of MySQL 4.1.1.

- `YEAR(`*`date`*`)`

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

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

- `YEARWEEK(`*`date`*`)`

  `YEARWEEK(`*`date`*`,`*`start`*`)`

  Returns year and week for a date. The *`start`* argument works exactly like the *`start`* argument to `WEEK()`. The year in the result may be different from the year in the date argument for the first and the last week of the year.

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

  Note that the week number is different from what the `WEEK()` function would return (`0`) for optional arguments `0` or `1`, as `WEEK()` then returns the week in the context of the given year.

  `YEARWEEK()` was added in MySQL 3.23.8.

## 5.6 Full-Text Search Functions

- `MATCH (`*`col1`*`,`*`col2`*`,...) AGAINST (`*`expr`*` [IN BOOLEAN MODE | WITH QUERY EXPANSION])`

  As of MySQL 3.23.23, MySQL has support for full-text indexing and searching. A full-text index in MySQL is an index of type `FULLTEXT`. `FULLTEXT` indexes are used with `MyISAM` tables only and can be created from `CHAR`, `VARCHAR`, or `TEXT` columns at `CREATE TABLE` time or added later with `ALTER TABLE` or `CREATE INDEX`. For large datasets, it will be much faster to load your data into a table that has no `FULLTEXT` index, then create the index with `ALTER TABLE` (or `CREATE INDEX`). Loading data into a table that already has a `FULLTEXT` index could be significantly slower.

  Constraints on full-text searching are listed in Section 5.6.3, "Full-Text Restrictions."

Full-text searching is performed with the `MATCH()` function.

```
mysql> CREATE TABLE articles (
    ->   id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
    ->   title VARCHAR(200),
    ->   body TEXT,
    ->   FULLTEXT (title,body)
    -> );
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> INSERT INTO articles (title,body) VALUES
    -> ('MySQL Tutorial','DBMS stands for DataBase ...'),
    -> ('How To Use MySQL Well','After you went through a ...'),
    -> ('Optimizing MySQL','In this tutorial we will show ...'),
    -> ('1001 MySQL Tricks','1. Never run mysqld as root. 2. ...'),
    -> ('MySQL vs. YourSQL','In the following database comparison ...'),
    -> ('MySQL Security','When configured properly, MySQL ...');
Query OK, 6 rows affected (0.00 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM articles
    -> WHERE MATCH (title,body) AGAINST ('database');
+----+-------------------+------------------------------------------+
| id | title             | body                                     |
+----+-------------------+------------------------------------------+
|  5 | MySQL vs. YourSQL | In the following database comparison ... |
|  1 | MySQL Tutorial    | DBMS stands for DataBase ...             |
+----+-------------------+------------------------------------------+
2 rows in set (0.00 sec)
```

The `MATCH()` function performs a natural language search for a string against a text collection. A collection is a set of one or more columns included in a `FULLTEXT` index. The search string is given as the argument to `AGAINST()`. The search is performed in case-insensitive fashion. For every row in the table, `MATCH()` returns a relevance value, that is, a similarity measure between the search string and the text in that row in the columns named in the `MATCH()` list.

When `MATCH()` is used in a `WHERE` clause, as in the preceding example, the rows returned are automatically sorted with the highest relevance first. Relevance values are non-negative floating-point numbers. Zero relevance means no similarity. Relevance is computed based on the number of words in the row, the number of unique words in that row, the total number of words in the collection, and the number of documents (rows) that contain a particular word.

For natural-language full-text searches, it is a requirement that the columns named in the `MATCH()` function be the same columns included in some `FULLTEXT` index in your table. For the preceding query, note that the columns named in the `MATCH()` function (`title` and `body`) are the same as those named in the definition of the `article` table's `FULLTEXT` index. If you wanted to search the `title` or `body` separately, you would need to create `FULLTEXT` indexes for each column.

It is also possible to perform a boolean search or a search with query expansion. These search types are described in Section 5.6.1, "Boolean Full-Text Searches," and Section 5.6.2, "Full-Text Searches with Query Expansion."

The preceding example is a basic illustration showing how to use the `MATCH()` function where rows are returned in order of decreasing relevance. The next example shows how to retrieve the relevance values explicitly. Returned rows are not ordered because the SELECT statement includes neither WHERE nor ORDER BY clauses:

```
mysql> SELECT id, MATCH (title,body) AGAINST ('Tutorial')
    -> FROM articles;
+----+-----------------------------------------+
| id | MATCH (title,body) AGAINST ('Tutorial') |
+----+-----------------------------------------+
|  1 |                        0.65545833110809 |
|  2 |                                       0 |
|  3 |                        0.66266459226608 |
|  4 |                                       0 |
|  5 |                                       0 |
|  6 |                                       0 |
+----+-----------------------------------------+
6 rows in set (0.00 sec)
```

The following example is more complex. The query returns the relevance values and it also sorts the rows in order of decreasing relevance. To achieve this result, you should specify MATCH() twice: once in the SELECT list and once in the WHERE clause. This causes no additional overhead, because the MySQL optimizer notices that the two MATCH() calls are identical and invokes the full-text search code only once.

```
mysql> SELECT id, body, MATCH (title,body) AGAINST
    -> ('Security implications of running MySQL as root') AS score
    -> FROM articles WHERE MATCH (title,body) AGAINST
    -> ('Security implications of running MySQL as root');
+----+-------------------------------------+-----------------+
| id | body                                | score           |
+----+-------------------------------------+-----------------+
|  4 | 1. Never run mysqld as root. 2. ... | 1.5219271183014 |
|  6 | When configured properly, MySQL ... | 1.3114095926285 |
+----+-------------------------------------+-----------------+
2 rows in set (0.00 sec)
```

MySQL uses a very simple parser to split text into words. A "word" is any sequence of characters consisting of letters, digits, '''', or '_'. Some words are ignored in full-text searches:

- Any word that is too short is ignored. The default minimum length of words that will be found by full-text searches is four characters.
- Words in the stopword list are ignored. A stopword is a word such as "the" or "some" that is so common that it is considered to have zero semantic value. There is a built-in stopword list.

The default minimum word length and stopword list can be changed as described in Section 5.6.4, "Fine-Tuning MySQL Full-Text Search."

Every correct word in the collection and in the query is weighted according to its significance in the collection or query. This way, a word that is present in many documents has a lower weight (and may even have a zero weight), because it has lower semantic value in this particular collection. Conversely, if the word is rare, it receives a higher weight. The weights of the words are then combined to compute the relevance of the row.

Such a technique works best with large collections (in fact, it was carefully tuned this way). For very small tables, word distribution does not adequately reflect their semantic value, and this model may sometimes produce bizarre results. For example, although the word "MySQL" is present in every row of the `articles` table, a search for the word produces no results:

```
mysql> SELECT * FROM articles
    -> WHERE MATCH (title,body) AGAINST ('MySQL');
Empty set (0.00 sec)
```

The search result is empty because the word "MySQL" is present in at least 50% of the rows. As such, it is effectively treated as a stopword. For large datasets, this is the most desirable behavior—a natural language query should not return every second row from a 1GB table. For small datasets, it may be less desirable.

A word that matches half of the rows in a table is less likely to locate relevant documents. In fact, it will most likely find plenty of irrelevant documents. We all know this happens far too often when we are trying to find something on the Internet with a search engine. It is with this reasoning that rows containing the word are assigned a low semantic value for *the particular dataset in which they occur*. A given word may exceed the 50% threshold in one dataset but not another.

The 50% threshold has a significant implication when you first try full-text searching to see how it works: If you create a table and insert only one or two rows of text into it, every word in the text occurs in at least 50% of the rows. As a result, no search returns any results. Be sure to insert at least three rows, and preferably many more.

## 5.6.1 Boolean Full-Text Searches

As of Version 4.0.1, MySQL can also perform boolean full-text searches using the `IN BOOLEAN MODE` modifier.

```
mysql> SELECT * FROM articles WHERE MATCH (title,body)
    -> AGAINST ('+MySQL -YourSQL' IN BOOLEAN MODE);
+----+-----------------------+-------------------------------------+
| id | title                 | body                                |
+----+-----------------------+-------------------------------------+
|  1 | MySQL Tutorial        | DBMS stands for DataBase ...        |
|  2 | How To Use MySQL Well | After you went through a ...        |
|  3 | Optimizing MySQL      | In this tutorial we will show ...   |
|  4 | 1001 MySQL Tricks     | 1. Never run mysqld as root. 2. ... |
|  6 | MySQL Security        | When configured properly, MySQL ... |
+----+-----------------------+-------------------------------------+
```

This query retrieves all the rows that contain the word "MySQL" but that do *not* contain the word "YourSQL".

Boolean full-text searches have these characteristics:

- They do not use the 50% threshold.
- They do not automatically sort rows in order of decreasing relevance. You can see this from the preceding query result: The row with the highest relevance is the one that contains "MySQL" twice, but it is listed last, not first.
- They can work even without a `FULLTEXT` index, although this would be *slow*.

The boolean full-text search capability supports the following operators:

- `+`

  A leading plus sign indicates that this word *must be* present in every row returned.
- `-`

  A leading minus sign indicates that this word *must not be* present in any row returned.
- `(no operator)`

  By default (when neither `+` nor `-` is specified) the word is optional, but the rows that contain it will be rated higher. This mimics the behavior of `MATCH() ... AGAINST()` without the `IN BOOLEAN MODE` modifier.
- `> <`

  These two operators are used to change a word's contribution to the relevance value that is assigned to a row. The `>` operator increases the contribution and the `<` operator decreases it. See the example below.
- `( )`

  Parentheses are used to group words into subexpressions. Parenthesized groups can be nested.
- `~`

  A leading tilde acts as a negation operator, causing the word's contribution to the row relevance to be negative. It's useful for marking noise words. A row that contains such a word will be rated lower than others, but will not be excluded altogether, as it would be with the `-` operator.
- `*`

  An asterisk is the truncation operator. Unlike the other operators, it should be *appended* to the word.
- `"`

  A phrase that is enclosed within double quote ('`"`') characters matches only rows that contain the phrase *literally*, *as it was typed*.

The following examples demonstrate some search strings that use boolean full-text operators:

- `'apple banana'`

  Find rows that contain at least one of the two words.
- `'+apple +juice'`

  Find rows that contain both words.
- `'+apple macintosh'`

  Find rows that contain the word "apple", but rank rows higher if they also contain "macintosh".
- `'+apple -macintosh'`

  Find rows that contain the word "apple" but not "macintosh".
- `'+apple +(>turnover <strudel)'`

  Find rows that contain the words "apple" and "turnover", or "apple" and "strudel" (in any order), but rank "apple turnover" higher than "apple strudel".
- `'apple*'`

  Find rows that contain words such as "apple", "apples", "applesauce", or "applet".
- `'"some words"'`

  Find rows that contain the exact phrase "some words" (for example, rows that contain "some words of wisdom" but not "some noise words"). Note that the '`"`' characters that surround the phrase are operator characters that delimit the phrase. They are not the quotes that surround the search string itself.

## 5.6.2 Full-Text Searches with Query Expansion

As of MySQL 4.1.1, full-text search supports query expansion (in particular, its variant "blind query expansion"). This is generally useful when a search phrase is too short, which often means that the user is relying on implied knowledge that the full-text search engine usually lacks. For example, a user searching for "database" may really mean that "MySQL", "Oracle", "DB2", and "RDBMS" all are phrases that should match "databases" and should be returned, too. This is implied knowledge.

Blind query expansion (also known as automatic relevance feedback) is enabled by adding `WITH QUERY EXPANSION` following the search phrase. It works by performing the search twice, where the search phrase for the second search is the original search phrase concatenated with the few top found documents from the first search. Thus, if one of these documents contains the word "databases" and the word "MySQL", the second search will find the documents that contain the word "MySQL" even if they do not contain the word "database". The following example shows this difference:

```
mysql> SELECT * FROM articles
    -> WHERE MATCH (title,body) AGAINST ('database');
+----+-------------------+------------------------------------------+
| id | title             | body                                     |
+----+-------------------+------------------------------------------+
|  5 | MySQL vs. YourSQL | In the following database comparison ... |
|  1 | MySQL Tutorial    | DBMS stands for DataBase ...             |
+----+-------------------+------------------------------------------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM articles
    -> WHERE MATCH (title,body)
    -> AGAINST ('database' WITH QUERY EXPANSION);
+----+-------------------+------------------------------------------+
| id | title             | body                                     |
+----+-------------------+------------------------------------------+
|  1 | MySQL Tutorial    | DBMS stands for DataBase ...             |
|  5 | MySQL vs. YourSQL | In the following database comparison ... |
|  3 | Optimizing MySQL  | In this tutorial we will show ...        |
+----+-------------------+------------------------------------------+
3 rows in set (0.00 sec)
```

Another example could be searching for books by Georges Simenon about Maigret, when a user is not sure how to spell "Maigret". A search for "Megre and the reluctant witnesses" will find only "Maigret and the Reluctant Witnesses" without query expansion. A search with query expansion will find all books with the word "Maigret" on the second pass.

**Note:** Because blind query expansion tends to increase noise significantly by returning non-relevant documents, it's only meaningful to use when a search phrase is rather short.

## 5.6.3 Full-Text Restrictions

- Full-text searches are supported for `MyISAM` tables only.
- As of MySQL 4.1.1, full-text searches can be used with most multi-byte character sets. The exception is that for Unicode, the `utf8` character set can be used, but not the `ucs2` character set.
- As of MySQL 4.1, the use of multiple character sets within a single table is supported. However, all columns in a `FULLTEXT` index must have the same character set and collation.
- The `MATCH()` column list must exactly match the column list in some `FULLTEXT` index definition for the table, unless this `MATCH()` is `IN BOOLEAN MODE`.
- The argument to `AGAINST()` must be a constant string.

## 5.6.4 Fine-Tuning MySQL Full-Text Search

The MySQL full-text search capability has few user-tunable parameters yet, although adding more is very high on the TODO. You can exert more control over full-text searching behavior if you have a MySQL source distribution because some changes require source code modifications.

Note that full-text search was carefully tuned for the best searching effectiveness. Modifying the default behavior will, in most cases, make the search results worse. Do not alter the MySQL sources unless you know what you are doing!

Most full-text variables described in the following items must be set at server startup time. For these variables, a server restart is required to change them and you cannot modify them dynamically while the server is running.

Some variable changes require that you rebuild the `FULLTEXT` indexes in your tables. Instructions for doing this are given at the end of this section.

- The minimum and maximum length of words to be indexed is defined by the `ft_min_word_len` and `ft_max_word_len` system variables (available as of MySQL 4.0.0). The default minimum value is four characters. The default maximum depends on your version of MySQL. If you change either value, you must rebuild your `FULLTEXT` indexes. For example, if you want three-character words to be searchable, you can set the `ft_min_word_len` variable by putting the following lines in an option file:

  ```
  [mysqld]
  ft_min_word_len=3
  ```

  Then restart the server and rebuild your `FULLTEXT` indexes. Also note particularly the remarks regarding `myisamchk` in the instructions following this list.
- To override the default stopword list, set the `ft_stopword_file` system variable (available as of MySQL 4.0.10). The variable value should be the pathname of the file containing the stopword list, or the empty string to disable stopword filtering. After changing the value, rebuild your `FULLTEXT` indexes.
- The 50% threshold for natural language searches is determined by the particular weighting scheme chosen. To disable it, look for the following line in `myisam/ftdefs.h`:

  ```
  #define GWS_IN_USE GWS_PROB
  ```

  Change the line to this:

  ```
  #define GWS_IN_USE GWS_FREQ
  ```

  Then recompile MySQL. There is no need to rebuild the indexes in this case. **Note:** By doing this you *severely* decrease MySQL's capability to provide adequate relevance values for the `MATCH()` function. If you really need to search for such common words, it would be better to search using `IN BOOLEAN MODE` instead, which does not observe the 50% threshold.

- To change the operators used for boolean full-text searches, set the ft_boolean_syntax system variable (available as of MySQL 4.0.1). The variable also can be changed while the server is running, but you must have the SUPER privilege to do so. No index rebuilding is necessary.

If you modify full-text variables that affect indexing (ft_min_word_len, ft_max_word_len, or ft_stopword_file), you must rebuild your FULLTEXT indexes after making the changes and restarting the server. To rebuild the indexes in this case, it's sufficient to do a QUICK repair operation:

```
mysql> REPAIR TABLE tbl_name QUICK;
```

With regard specifically to using the IN BOOLEAN MODE capability, if you upgrade from MySQL 3.23 to 4.0 or later, it's necessary to replace the index header as well. To do this, do a USE_FRM repair operation:

```
mysql> REPAIR TABLE tbl_name USE_FRM;
```

This is necessary because boolean full-text searches require a flag in the index header that was not present in MySQL 3.23, and that is not added if you do only a QUICK repair. If you attempt a boolean full-text search without rebuilding the indexes this way, the search will return incorrect results.

Note that if you use myisamchk to perform an operation that modifies table indexes (such as repair or analyze), the FULLTEXT indexes are rebuilt using the default full-text parameter values for minimum and maximum word length and the stopword file unless you specify otherwise. This can result in queries failing.

The problem occurs because these parameters are known only by the server. They are not stored in MyISAM index files. To avoid the problem if you have modified the minimum or maximum word length or the stopword file in the server, specify the same ft_min_word_len, ft_max_word_len, and ft_stopword_file values to myisamchk that you use for mysqld. For example, if you have set the minimum word length to 3, you can repair a table with myisamchk like this:

```
shell> myisamchk --recover --ft_min_word_len=3 tbl_name.MYI
```

To ensure that myisamchk and the server use the same values for full-text parameters, you can place each one in both the [mysqld] and [myisamchk] sections of an option file:

```
[mysqld]
ft_min_word_len=3

[myisamchk]
ft_min_word_len=3
```

An alternative to using myisamchk is to use the REPAIR TABLE, ANALYZE TABLE, OPTIMIZE TABLE, or ALTER TABLE. These statements are performed by the server, which knows the proper full-text parameter values to use.

### 5.6.5 Full-Text Search TODO

- Improved performance for all `FULLTEXT` operations.
- Proximity operators.
- Support for "always-index words." These could be any strings the user wants to treat as words, such as "C++", "AS/400", or "TCP/IP".
- Support for full-text search in `MERGE` tables.
- Support for the `ucs2` character set.
- Make the stopword list dependent on the language of the dataset.
- Stemming (dependent on the language of the dataset).
- Generic user-suppliable UDF preparser.
- Make the model more flexible (by adding some adjustable parameters to `FULLTEXT` in `CREATE TABLE` and `ALTER TABLE` statements).

## 5.7 Cast Functions

- `CAST(`*`expr`*` AS `*`type`*`)`

  `CONVERT(`*`expr`*`,`*`type`*`)`

  `CONVERT(`*`expr`*` USING `*`transcoding_name`*`)`

  The `CAST()` and `CONVERT()` functions may be used to take a value of one type and produce a value of another type.

  The *`type`* can be one of the following values:

  - `BINARY`
  - `CHAR`
  - `DATE`
  - `DATETIME`
  - `SIGNED [INTEGER]`
  - `TIME`
  - `UNSIGNED [INTEGER]`

  `CAST()` and `CONVERT()` are available as of MySQL 4.0.2. The `CHAR` conversion type is available as of 4.0.6. The `USING` form of `CONVERT()` is available as of 4.1.0.

  `CAST()` and `CONVERT(... USING ...)` are standard SQL syntax. The non-`USING` form of `CONVERT()` is ODBC syntax.

> `CONVERT()` with `USING` is used to convert data between different character sets. In MySQL, transcoding names are the same as the corresponding character set names. For example, this statement converts the string `'abc'` in the server's default character set to the corresponding string in the `utf8` character set:
>
> ```
> SELECT CONVERT('abc' USING utf8);
> ```

The cast functions are useful when you want to create a column with a specific type in a `CREATE ... SELECT` statement:

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

The functions also can be useful for sorting `ENUM` columns in lexical order. Normally sorting of `ENUM` columns occurs using the internal numeric values. Casting the values to `CHAR` results in a lexical sort:

```
SELECT enum_col FROM tbl_name ORDER BY CAST(enum_col AS CHAR);
```

`CAST(`*`str`*` AS BINARY)` is the same thing as `BINARY `*`str`*. `CAST(`*`expr`*` AS CHAR)` treats the expression as a string with the default character set.

**Note:** In MysQL 4.0, a `CAST()` to `DATE`, `DATETIME`, or `TIME` only marks the column to be a specific type but doesn't change the value of the column.

As of MySQL 4.1.0, the value is converted to the correct column type when it's sent to the user (this is a feature of how the new protocol in 4.1 sends date information to the client):

```
mysql> SELECT CAST(NOW() AS DATE);
       -> 2003-05-26
```

As of MySQL 4.1.1, `CAST()` also changes the result if you use it as part of a more complex expression such as `CONCAT('Date: ',CAST(NOW() AS DATE))`.

You should not use `CAST()` to extract data in different formats but instead use string functions like `LEFT()` or `EXTRACT()`. See Section 5.5, "Date and Time Functions."

To cast a string to a numeric value, you don't normally have to do anything. Just use the string value as though it were a number:

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

If you use a number in string context, the number automatically is converted to a `BINARY` string.

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

MySQL supports arithmetic with both signed and unsigned 64-bit values. If you are using numerical operators (such as +) and one of the operands is an unsigned integer, the result is unsigned. You can override this by using the `SIGNED` and `UNSIGNED` cast operators to cast the operation to a signed or unsigned 64-bit integer, respectively.

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

Note that if either operand is a floating-point value, the result is a floating-point value and is not affected by the preceding rule. (In this context, `DECIMAL` column values are regarded as floating-point values.)

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

If you are using a string in an arithmetic operation, this is converted to a floating-point number.

The handing of unsigned values was changed in MySQL 4.0 to be able to support `BIGINT` values properly. If you have some code that you want to run in both MySQL 4.0 and 3.23, you probably can't use the `CAST()` function. You can use the following technique to get a signed result when subtracting two unsigned integer columns `ucol1` and `ucol2`:

```
mysql> SELECT (ucol1+0.0)-(ucol2+0.0) FROM ...;
```

The idea is that the columns are converted to floating-point values before the subtraction occurs.

If you have a problem with `UNSIGNED` columns in old MySQL applications when porting them to MySQL 4.0, you can use the `--sql-mode=NO_UNSIGNED_SUBTRACTION` option when starting `mysqld`. However, as long as you use this option, you will not be able to make efficient use of the `BIGINT UNSIGNED` column type.

## 5.8 Other Functions

### 5.8.1 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
  ```

  The result is an unsigned 64-bit integer.

- `&`

  Bitwise AND:

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

  The result is an unsigned 64-bit integer.

- `^`

  Bitwise XOR:

  ```
  mysql> SELECT 1 ^ 1;
          -> 0
  mysql> SELECT 1 ^ 0;
          -> 1
  mysql> SELECT 11 ^ 3;
          -> 8
  ```

  The result is an unsigned 64-bit integer.

  Bitwise XOR was added in MySQL 4.0.2.

- `<<`

  Shifts a longlong (`BIGINT`) number to the left.

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

  The result is an unsigned 64-bit integer.

- `>>`

  Shifts a longlong (`BIGINT`) number to the right.

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

  The result is an unsigned 64-bit integer.

- `~`

  Invert all bits.

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

  The result is an unsigned 64-bit integer.

- `BIT_COUNT(N)`

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

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

## 5.8.2 Encryption Functions

The functions in this section encrypt and decrypt data values. If you want to store results from an encryption function that might contain arbitrary byte values, use a `BLOB` column rather than a `CHAR` or `VARCHAR` column to avoid potential problems with trailing space removal that would change data values.

- `AES_ENCRYPT(`*`str`*`,`*`key_str`*`)`

  `AES_DECRYPT(`*`crypt_str`*`,`*`key_str`*`)`

  These functions allow encryption and decryption of data using the official AES (Advanced Encryption Standard) algorithm, previously known as "Rijndael." Encoding with a 128-bit key length is used, but you can extend it up to 256 bits by modifying the source. We chose 128 bits because it is much faster and it is usually secure enough.

  The input arguments may be any length. If either argument is `NULL`, the result of this function is also `NULL`.

  Because AES is a block-level algorithm, padding is used to encode uneven length strings and so the result string length may be calculated as `16*(trunc(`*`string_length`*`/16)+1)`.

  If `AES_DECRYPT()` detects invalid data or incorrect padding, it returns `NULL`. However, it is possible for `AES_DECRYPT()` to return a non-`NULL` value (possibly garbage) if the input data or the key is invalid.

  You can use the AES functions to store data in an encrypted form by modifying your queries:

  ```
  INSERT INTO t VALUES (1,AES_ENCRYPT('text','password'));
  ```

  You can get even more security by not transferring the key over the connection for each query, which can be accomplished by storing it in a server-side variable at connection time. For example:

  ```
  SELECT @password:='my password';
  INSERT INTO t VALUES (1,AES_ENCRYPT('text',@password));
  ```

  `AES_ENCRYPT()` and `AES_DECRYPT()` were added in MySQL 4.0.2, and can be considered the most cryptographically secure encryption functions currently available in MySQL.

- `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()`.

- `ENCODE(`*`str`*`,`*`pass_str`*`)`

  Encrypt *`str`* using *`pass_str`* as the password. To decrypt the result, use `DECODE()`.

  The result is a binary string of the same length as *`str`*. If you want to save it in a column, use a `BLOB` column type.

- `DES_DECRYPT(`*`crypt_str`*`[,`*`key_str`*`])`

  Decrypts a string encrypted with `DES_ENCRYPT()`. On error, this function returns `NULL`.

  Note that this function works only if MySQL has been configured with SSL support.

  If no *`key_str`* argument is given, `DES_DECRYPT()` examines the first byte of the encrypted string to determine the DES key number that was used to encrypt the original string, and then reads the key from the DES key file to decrypt the message. For this to work, the user must have the `SUPER` privilege. The key file can be specified with the `--des-key-file` server option.

  If you pass this function a *`key_str`* argument, that string is used as the key for decrypting the message.

  If the *`crypt_str`* argument doesn't look like an encrypted string, MySQL will return the given *`crypt_str`*.

  `DES_DECRYPT()` was added in MySQL 4.0.1.

- `DES_ENCRYPT(`*`str`*`[,(`*`key_num`*`|`*`key_str`*`)])`

  Encrypts the string with the given key using the Triple-DES algorithm. On error, this function returns `NULL`.

  Note that this function works only if MySQL has been configured with SSL support.

  The encryption key to use is chosen based on the second argument to `DES_ENCRYPT()`, if one was given:

  | Argument | Description |
  |---|---|
  | No argument | The first key from the DES key file is used. |
  | *key_num* | The given key number (0-9) from the DES key file is used. |
  | *key_str* | The given key string is used to encrypt *str*. |

  The key file can be specified with the `--des-key-file` server option.

  The return string is a binary string where the first character is `CHAR(128|`*`key_num`*`)`.

  The 128 is added to make it easier to recognize an encrypted key. If you use a string key, *`key_num`* will be 127.

  The string length for the result will be *`new_len`* `=` *`orig_len`* `+ (8-(`*`orig_len`*` % 8))+1`.

  Each line in the DES key file has the following format:

  ```
  key_num des_key_str
  ```

  Each *`key_num`* must be a number in the range from `0` to `9`. Lines in the file may be in any order. *`des_key_str`* is the string that will be used to encrypt the message. Between the number and the key there should be at least one space. The first key is the default key that is used if you don't specify any key argument to `DES_ENCRYPT()`.

You can tell MySQL to read new key values from the key file with the `FLUSH DES_KEY_FILE` statement. This requires the `RELOAD` privilege.

One benefit of having a set of default keys is that it gives applications a way to check for the existence of encrypted column values, without giving the end user the right to decrypt those values.

```
mysql> SELECT customer_address FROM customer_table WHERE
    -> crypted_credit_card = DES_ENCRYPT('credit_card_number');
```

`DES_DECRYPT()` was added in MySQL 4.0.1.

- `ENCRYPT(`*`str`*`[,`*`salt`*`])`

  Encrypt *`str`* using the Unix `crypt()` system call. The *`salt`* argument should be a string with two characters. (As of MySQL 3.22.16, *`salt`* may be longer than two characters.)

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

  `ENCRYPT()` ignores all but the first eight characters of *`str`*, at least on some systems. This behavior is determined by the implementation of the underlying `crypt()` system call.

  If `crypt()` is not available on your system, `ENCRYPT()` always returns `NULL`. Because of this, we recommend that you use `MD5()` or `SHA1()` instead, because those two functions exist on all platforms.

- `MD5(`*`str`*`)`

  Calculates an MD5 128-bit checksum for the string. The value is returned as a string of 32 hex digits, or `NULL` if the argument was `NULL`. The return value can, for example, be used as a hash key.

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

  This is the “RSA Data Security, Inc. MD5 Message-Digest Algorithm.”

  `MD5()` was added in MySQL 3.23.2.

- `OLD_PASSWORD(`*`str`*`)`

  `OLD_PASSWORD()` is available as of MySQL 4.1, when the implementation of `PASSWORD()` was changed to improve security. `OLD_PASSWORD()` returns the value of the pre-4.1 implementation of `PASSWORD()`.

- `PASSWORD(`*`str`*`)`

  Calculates and returns a password string from the plaintext password *`str`*, or `NULL` if the argument was `NULL`. 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 one-way (not reversible).

PASSWORD() does not perform password encryption in the same way that Unix passwords are encrypted. See ENCRYPT().

**Note:** The PASSWORD() function is used by the authentication system in MySQL Server, you should *not* use it in your own applications. For that purpose, use MD5() or SHA1() instead. Also see RFC 2195 for more information about handling passwords and authentication securely in your application.

- SHA1(*str*)

  SHA(*str*)

  Calculates an SHA1 160-bit checksum for the string, as described in RFC 3174 (Secure Hash Algorithm). The value is returned as a string of 40 hex digits, or NULL if the argument was NULL. One of the possible uses for this function is as a hash key. You can also use it as a cryptographically safe function for storing passwords.

```
mysql> SELECT SHA1('abc');
        -> 'a9993e364706816aba3e25717850c26c9cd0d89d'
```

  SHA1() was added in MySQL 4.0.2, and can be considered a cryptographically more secure equivalent of MD5(). SHA() is synonym for SHA1().

## 5.8.3 Information Functions

- 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 from within 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 is advisable to execute BENCHMARK() several times, and to interpret the result with regard to how heavily loaded the server machine is.

- `CHARSET(str)`

  Returns the character set of the string argument.

  ```
  mysql> SELECT CHARSET('abc');
          -> 'latin1'
  mysql> SELECT CHARSET(CONVERT('abc' USING utf8));
          -> 'utf8'
  mysql> SELECT CHARSET(USER());
          -> 'utf8'
  ```

  `CHARSET()` was added in MySQL 4.1.0.

- `COERCIBILITY(str)`

  Returns the collation coercibility value of the string argument.

  ```
  mysql> SELECT COERCIBILITY('abc' COLLATE latin1_swedish_ci);
          -> 0
  mysql> SELECT COERCIBILITY('abc');
          -> 3
  mysql> SELECT COERCIBILITY(USER());
          -> 2
  ```

  The return values have the following meanings:

  | Coercibility | Meaning |
  |---|---|
  | 0 | Explicit collation |
  | 1 | No collation |
  | 2 | Implicit collation |
  | 3 | Coercible |

  Lower values have higher precedence.

  `COERCIBILITY()` was added in MySQL 4.1.1.

- `COLLATION(str)`

  Returns the collation for the character set of the string argument.

  ```
  mysql> SELECT COLLATION('abc');
          -> 'latin1_swedish_ci'
  mysql> SELECT COLLATION(_utf8'abc');
          -> 'utf8_general_ci'
  ```

  `COLLATION()` was added in MySQL 4.1.0.

- `CONNECTION_ID()`

  Returns the connection ID (thread ID) for the connection. Every connection has its own unique ID.

  ```
  mysql> SELECT CONNECTION_ID();
          -> 23786
  ```

  `CONNECTION_ID()` was added in MySQL 3.23.14.

- `CURRENT_USER()`

  Returns the username and hostname combination that the current session was authenticated as. This value corresponds to the MySQL account that determines your access privileges. It can be different from the value of `USER()`.

  ```
  mysql> SELECT USER();
          -> 'davida@localhost'
  mysql> SELECT * FROM mysql.user;
  ERROR 1044: Access denied for user: '@localhost' to
  database 'mysql'
  mysql> SELECT CURRENT_USER();
          -> '@localhost'
  ```

  The example illustrates that although the client specified a username of `davida` (as indicated by the value of the `USER()` function), the server authenticated the client using an anonymous user account (as seen by the empty username part of the `CURRENT_USER()` value). One way this might occur is that there is no account listed in the grant tables for `davida`.

  `CURRENT_USER()` was added in MySQL 4.0.6.

- `DATABASE()`

  Returns the default (current) database name.

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

  If there is no default database, `DATABASE()` returns `NULL` as of MySQL 4.1.1, and the empty string before that.

- `FOUND_ROWS()`

  A `SELECT` statement may include a `LIMIT` clause to restrict the number of rows the server returns to the client. In some cases, it is desirable to know how many rows the statement would have returned without the `LIMIT`, but without running the statement again. To get this row count, include a `SQL_CALC_FOUND_ROWS` option in the `SELECT` statement, then invoke `FOUND_ROWS()` afterward:

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

  The second `SELECT` will return a number indicating how many rows the first `SELECT` would have returned had it been written without the `LIMIT` clause. (If the preceding `SELECT` statement does not include the `SQL_CALC_FOUND_ROWS` option, then `FOUND_ROWS()` may return a different result when `LIMIT` is used than when it is not.)

  Note that if you are using `SELECT SQL_CALC_FOUND_ROWS`, MySQL must calculate how many rows are in the full result set. However, this is faster than running the query again without `LIMIT`, because the result set need not be sent to the client.

`SQL_CALC_FOUND_ROWS` and `FOUND_ROWS()` can be useful in situations when you want to restrict the number of rows that a query returns, but also determine the number of rows in the full result set without running the query again. An example is a Web script that presents a paged display containing links to the pages that show other sections of a search result. Using `FOUND_ROWS()` allows you to determine how many other pages are needed for the rest of the result.

The use of `SQL_CALC_FOUND_ROWS` and `FOUND_ROWS()` is more complex for `UNION` queries than for simple `SELECT` statements, because `LIMIT` may occur at multiple places in a `UNION`. It may be applied to individual `SELECT` statements in the `UNION`, or global to the `UNION` result as a whole.

The intent of `SQL_CALC_FOUND_ROWS` for `UNION` is that it should return the row count that would be returned without a global `LIMIT`. The conditions for use of `SQL_CALC_FOUND_ROWS` with `UNION` are:

- The `SQL_CALC_FOUND_ROWS` keyword must appear in the first `SELECT` of the `UNION`.
- The value of `FOUND_ROWS()` is exact only if `UNION ALL` is used. If `UNION` without `ALL` is used, duplicate removal occurs and the value of `FOUND_ROWS()` is only approximate.
- If no `LIMIT` is present in the `UNION`, `SQL_CALC_FOUND_ROWS` is ignored and returns the number of rows in the temporary table that is created to process the `UNION`.

`SQL_CALC_FOUND_ROWS` and `FOUND_ROWS()` are available starting at MySQL 4.0.0.

- `LAST_INSERT_ID()`

  `LAST_INSERT_ID(`*`expr`*`)`

  Returns the last automatically generated value that was inserted into an `AUTO_INCREMENT` column.

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

  The last ID that was generated is maintained in the server on a per-connection basis. This means the value the function returns to a given client is the most recent `AUTO_INCREMENT` value generated by that client. The value cannot be affected by other clients, even if they generate `AUTO_INCREMENT` values of their own. This behavior ensures that you can retrieve your own ID without concern for the activity of other clients, and without the need for locks or transactions.

  The value of `LAST_INSERT_ID()` is not changed if you update the `AUTO_INCREMENT` column of a row with a non-magic value (that is, a value that is not `NULL` and not `0`).

  If you insert many rows at the same time with an insert statement, `LAST_INSERT_ID()` returns the value for the first inserted row. The reason for this is to make it possible to easily reproduce the same `INSERT` statement against some other server.

  If *`expr`* is given as an argument to `LAST_INSERT_ID()`, the value of the argument is returned by the function and is remembered as the next value to be returned by `LAST_INSERT_ID()`. This can be used to simulate sequences:

- Create a table to hold the sequence counter and initialize it:

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

- Use the table to generate sequence numbers like this:

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

  The UPDATE statement increments the sequence counter and causes the next call to LAST_INSERT_ID() to return the updated value. The SELECT statement retrieves that value. The mysql_insert_id() C API function can also be used to get the value.

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. It is multi-user safe because multiple clients can issue the UPDATE statement and get their own sequence value with the SELECT statement (or mysql_insert_id()), without affecting or being affected by other clients that generate their own sequence values.

Note that mysql_insert_id() is only updated after INSERT and UPDATE statements, so you cannot use the C API function to retrieve the value for LAST_INSERT_ID(*expr*) after executing other SQL statements like SELECT or SET.

- SESSION_USER()

  SESSION_USER() is a synonym for USER().

- SYSTEM_USER()

  SYSTEM_USER() is a synonym for USER().

- USER()

  Returns the current MySQL username and hostname.

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

  The value indicates the username you specified when connecting to the server, and the client host from which you connected. The value can be different than that of CURRENT_USER().

  Prior to MySQL 3.22.11, the function value does not include the client hostname. You can extract just the username part, regardless of whether the value includes a hostname part, like this:

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

  As of MySQL 4.1, USER() returns a value in the utf8 character set, so you should also make sure that the '@' string literal is interpreted in that character set:

  ```
  mysql> SELECT SUBSTRING_INDEX(USER(),_utf8'@',1);
          -> 'davida'
  ```

- `VERSION()`

  Returns a string that indicates the MySQL server version.

```
mysql> SELECT VERSION();
        -> '4.1.2-alpha-log'
```

  Note that if your version string ends with `-log` this means that logging is enabled.

## 5.8.4 Miscellaneous Functions

- `FORMAT(X,D)`

  Formats the number *`X`* to a format like `'#,###,###.##'`, rounded to *`D`* decimals, and returns the result as a string. If *`D`* is `0`, the result will have no decimal point or fractional part.

```
mysql> SELECT FORMAT(12332.123456, 4);
        -> '12,332.1235'
mysql> SELECT FORMAT(12332.1,4);
        -> '12,332.1000'
mysql> SELECT FORMAT(12332.2,0);
        -> '12,332'
```

- `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 (for example, because another client has already locked the name), or `NULL` if an error occurred (such as running out of memory or the thread was killed with `mysqladmin kill`). If you have a lock obtained with `GET_LOCK()`, it is released when you execute `RELEASE_LOCK()`, execute a new `GET_LOCK()`, or your connection terminates (either normally or abnormally).

  This function can be used to implement application locks or to simulate record locks. Names are locked on a server-wide basis. If a name has been locked by one client, `GET_LOCK()` blocks any request by another client for a lock with the same name. This allows clients that agree on a given lock name to use the name to perform cooperative advisory locking.

```
mysql> SELECT GET_LOCK('lock1',10);
        -> 1
mysql> SELECT IS_FREE_LOCK('lock2');
        -> 1
mysql> SELECT GET_LOCK('lock2',10);
        -> 1
mysql> SELECT RELEASE_LOCK('lock2');
        -> 1
mysql> SELECT RELEASE_LOCK('lock1');
        -> NULL
```

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

- `INET_ATON(`*`expr`*`)`

  Given the dotted-quad representation of a network address as a string, returns an integer that represents the numeric value of the address. Addresses may be 4- or 8-byte addresses.

  ```
  mysql> SELECT INET_ATON('209.207.224.40');
          -> 3520061480
  ```

  The generated number is always in network byte order. For the example just shown, the number is calculated as $209*256^3 + 207*256^2 + 224*256 + 40$.

  As of MySQL 4.1.2, `INET_ATON()` also understands short-form IP addresses:

  ```
  mysql> SELECT INET_ATON('127.0.0.1'), INET_ATON('127.1');
          -> 2130706433, 2130706433
  ```

  `INET_ATON()` was added in MySQL 3.23.15.

- `INET_NTOA(`*`expr`*`)`

  Given a numeric network address (4 or 8 byte), returns the dotted-quad representation of the address as a string.

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

  `INET_NTOA()` was added in MySQL 3.23.15.

- `IS_FREE_LOCK(`*`str`*`)`

  Checks whether the lock named *`str`* is free to use (that is, not locked). Returns `1` if the lock is free (no one is using the lock), `0` if the lock is in use, and `NULL` on errors (such as incorrect arguments).

  `IS_FREE_LOCK()` was added in MySQL 4.0.2.

- `IS_USED_LOCK(`*`str`*`)`

  Checks whether the lock named *`str`* is in use (that is, locked). If so, it returns the connection identifier of the client that holds the lock. Otherwise, it returns `NULL`.

  `IS_USED_LOCK()` was added in MySQL 4.1.0.

- `MASTER_POS_WAIT(`*`log_name`*`,`*`log_pos`*`[,`*`timeout`*`])`

  This function is useful for control of master/slave synchronization. It blocks until the slave has read and applied all updates up to the specified position in the master log. The return value is the number of log events it had to wait for to get to the specified position. The function returns `NULL` if the slave's SQL thread is not started, the slave's master information is not initialized, the arguments are incorrect, or an error occurs. It returns `-1` if the timeout has been exceeded. If the slave is already past the specified position, the function returns immediately.

If a *timeout* value is specified, MASTER_POS_WAIT() stops waiting when *timeout* seconds have elapsed. *timeout* must be greater than 0; a zero or negative *timeout* means no timeout.

MASTER_POS_WAIT() was added in MySQL 3.23.32. The *timeout* argument was added in 4.0.10.

- 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.

  The DO statement is convenient to use with RELEASE_LOCK(). See Section 6.1.2, "DO Syntax."

- UUID()

  Returns a Universal Unique Identifier (UUID) generated according to "DCE 1.1: Remote Procedure Call" (Appendix A) CAE (Common Applications Environment) Specifications published by The Open Group in October 1997 (Document Number C706).

  A UUID is designed as a number that is globally unique in space and time. Two calls to UUID() are expected to generate two different values, even if these calls are performed on two separate computers that are not connected to each other.

  A UUID is a 128-bit number represented by a string of five hexadecimal numbers in aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee format:

  - The first three numbers are generated from a timestamp.
  - The fourth number preserves temporal uniqueness in case the timestamp value loses monotonicity (for example, due to daylight saving time).
  - The fifth number is an IEEE 802 node number that provides spatial uniqueness. A random number is substituted if the latter is not available (for example, because the host computer has no Ethernet card, or we do not know how to find the hardware address of an interface on your operating system). In this case, spatial uniqueness cannot be guaranteed. Nevertheless, a collision should have *very* low probability.

    Currently, the MAC address of an interface is taken into account only on FreeBSD and Linux. On other operating systems, MySQL uses a randomly generated 48-bit number.

```
mysql> SELECT UUID();
        -> '6ccd780c-baba-1026-9564-0040f4311e29'
```

  Note that UUID() does not yet work with replication.

  UUID() was added in MySQL 4.1.2.

## 5.9 Functions and Modifiers for Use with GROUP BY Clauses

### 5.9.1 GROUP BY (Aggregate) Functions

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

- AVG(*expr*)

  Returns the average value of *expr*.

  ```
  mysql> SELECT student_name, AVG(test_score)
      ->        FROM student
      ->        GROUP BY student_name;
  ```

- BIT_AND(*expr*)

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

  As of MySQL 4.0.17, this function returns 18446744073709551615 if there were no matching rows. (This is an unsigned BIGINT value with all bits set to 1.) Before 4.0.17, the function returns -1 if there were no matching rows.

- BIT_OR(*expr*)

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

  This function returns 0 if there were no matching rows.

- BIT_XOR(*expr*)

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

  This function returns 0 if there were no matching rows.

  This function is available as of MySQL 4.1.1.

- COUNT(*expr*)

  Returns a count of the number of non-NULL values in the 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 somewhat different in that it returns a count of the number of rows retrieved, whether or not they contain `NULL` values.

`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;
```

This optimization applies only to `MyISAM` and `ISAM` tables, because an exact record count is stored for these table types and can be accessed very quickly. For transactional storage engines (`InnoDB`, `BDB`), storing an exact row count is more problematic because multiple transactions may be occurring, each of which may affect the count.

- `COUNT(DISTINCT` *`expr`*`,[`*`expr`*`...])`

  Returns a count of the number of different non-`NULL` values.

  ```
  mysql> SELECT COUNT(DISTINCT results) FROM student;
  ```

  In MySQL, you can get the number of distinct expression combinations that don't contain `NULL` by giving a list of expressions. In standard SQL, you would have to do a concatenation of all expressions inside `COUNT(DISTINCT ...)`.

  `COUNT(DISTINCT ...)` was added in MySQL 3.23.2.

- `GROUP_CONCAT(`*`expr`*`)`

  This function returns a string result with the concatenated values from a group. The full syntax is as follows:

  ```
  GROUP_CONCAT([DISTINCT] expr [,expr ...]
               [ORDER BY {unsigned_integer | col_name | expr}
                   [ASC | DESC] [,col ...]]
               [SEPARATOR str_val])
  ```

  ```
  mysql> SELECT student_name,
      ->     GROUP_CONCAT(test_score)
      ->     FROM student
      ->     GROUP BY student_name;
  ```

  Or:

  ```
  mysql> SELECT student_name,
      ->     GROUP_CONCAT(DISTINCT test_score
      ->               ORDER BY test_score DESC SEPARATOR ' ')
      ->     FROM student
      ->     GROUP BY student_name;
  ```

In MySQL, you can get the concatenated values of expression combinations. You can eliminate duplicate values by using DISTINCT. If you want to sort values in the result, you should use ORDER BY clause. To sort in reverse order, add the DESC (descending) keyword to the name of the column you are sorting by in the ORDER BY clause. The default is ascending order; this may be specified explicitly using the ASC keyword. SEPARATOR is followed by the string value that should be inserted between values of result. The default is a comma (','). You can remove the separator altogether by specifying SEPARATOR ''.

You can set a maximum allowed length with the group_concat_max_len system variable. The syntax to do this at runtime is as follows, where *val* is an unsigned integer:

```
SET [SESSION | GLOBAL] group_concat_max_len = val;
```

If a maximum length has been set, the result is truncated to this maximum length.

**Note:** There are still some small limitations with GROUP_CONCAT() when it comes to using DISTINCT together with ORDER BY and using BLOB values.

GROUP_CONCAT() was added in MySQL 4.1.

- 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;
  ```

  For MIN(), MAX(), and other aggregate functions, MySQL currently compares ENUM and SET columns by their string value rather than by the string's relative position in the set. This differs from how ORDER BY compares them. This will be rectified.

- STD(*expr*)

  STDDEV(*expr*)

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

- SUM(*expr*)

  Returns the sum of *expr*. Note that if the return set has no rows, it returns NULL!

- VARIANCE(*expr*)

  Returns the standard variance of *expr* (considering rows as the whole population, not as a sample; so it has the number of rows as denominator). This is an extension to standard SQL, available only in MySQL 4.1 or later.

## 5.9.2 GROUP BY Modifiers

As of MySQL 4.1.1, the `GROUP BY` clause allows a `WITH ROLLUP` modifier that causes extra rows to be added to the summary output. These rows represent higher-level (or super-aggregate) summary operations. `ROLLUP` thus allows you to answer questions at multiple levels of analysis with a single query. It can be used, for example, to provide support for OLAP (Online Analytical Processing) operations.

As an illustration, suppose that a table named `sales` has `year`, `country`, `product`, and `profit` columns for recording sales profitability:

```
CREATE TABLE sales
(
    year    INT NOT NULL,
    country VARCHAR(20) NOT NULL,
    product VARCHAR(32) NOT NULL,
    profit  INT
);
```

The table's contents can be summarized per year with a simple `GROUP BY` like this:

```
mysql> SELECT year, SUM(profit) FROM sales GROUP BY year;
+------+-------------+
| year | SUM(profit) |
+------+-------------+
| 2000 |        4525 |
| 2001 |        3010 |
+------+-------------+
```

This output shows the total profit for each year, but if you also want to determine the total profit summed over all years, you must add up the individual values yourself or run an additional query.

Or you can use `ROLLUP`, which provides both levels of analysis with a single query. Adding a `WITH ROLLUP` modifier to the `GROUP BY` clause causes the query to produce another row that shows the grand total over all year values:

```
mysql> SELECT year, SUM(profit) FROM sales GROUP BY year WITH ROLLUP;
+------+-------------+
| year | SUM(profit) |
+------+-------------+
| 2000 |        4525 |
| 2001 |        3010 |
| NULL |        7535 |
+------+-------------+
```

The grand total super-aggregate line is identified by the value `NULL` in the `year` column.

`ROLLUP` has a more complex effect when there are multiple `GROUP BY` columns. In this case, each time there is a "break" (change in value) in any but the last grouping column, the query produces an extra super-aggregate summary row.

For example, without `ROLLUP`, a summary on the `sales` table based on `year`, `country`, and `product` might look like this:

```
mysql> SELECT year, country, product, SUM(profit)
    -> FROM sales
    -> GROUP BY year, country, product;
+------+---------+------------+-------------+
| year | country | product    | SUM(profit) |
+------+---------+------------+-------------+
| 2000 | Finland | Computer   |        1500 |
| 2000 | Finland | Phone      |         100 |
| 2000 | India   | Calculator |         150 |
| 2000 | India   | Computer   |        1200 |
| 2000 | USA     | Calculator |          75 |
| 2000 | USA     | Computer   |        1500 |
| 2001 | Finland | Phone      |          10 |
| 2001 | USA     | Calculator |          50 |
| 2001 | USA     | Computer   |        2700 |
| 2001 | USA     | TV         |         250 |
+------+---------+------------+-------------+
```

The output indicates summary values only at the year/country/product level of analysis. When `ROLLUP` is added, the query produces several extra rows:

```
mysql> SELECT year, country, product, SUM(profit)
    -> FROM sales
    -> GROUP BY year, country, product WITH ROLLUP;
+------+---------+------------+-------------+
| year | country | product    | SUM(profit) |
+------+---------+------------+-------------+
| 2000 | Finland | Computer   |        1500 |
| 2000 | Finland | Phone      |         100 |
| 2000 | Finland | NULL       |        1600 |
| 2000 | India   | Calculator |         150 |
| 2000 | India   | Computer   |        1200 |
| 2000 | India   | NULL       |        1350 |
| 2000 | USA     | Calculator |          75 |
| 2000 | USA     | Computer   |        1500 |
| 2000 | USA     | NULL       |        1575 |
| 2000 | NULL    | NULL       |        4525 |
| 2001 | Finland | Phone      |          10 |
| 2001 | Finland | NULL       |          10 |
| 2001 | USA     | Calculator |          50 |
| 2001 | USA     | Computer   |        2700 |
| 2001 | USA     | TV         |         250 |
| 2001 | USA     | NULL       |        3000 |
| 2001 | NULL    | NULL       |        3010 |
| NULL | NULL    | NULL       |        7535 |
+------+---------+------------+-------------+
```

For this query, adding ROLLUP causes the output to include summary information at four levels of analysis, not just one. Here's how to interpret the ROLLUP output:

- Following each set of product rows for a given year and country, an extra summary row is produced showing the total for all products. These rows have the product column set to NULL.
- Following each set of rows for a given year, an extra summary row is produced showing the total for all countries and products. These rows have the country and products columns set to NULL.
- Finally, following all other rows, an extra summary row is produced showing the grand total for all years, countries, and products. This row has the year, country, and products columns set to NULL.

## Other Considerations When Using ROLLUP

The following items list some behaviors specific to the MySQL implementation of ROLLUP:

When you use ROLLUP, you cannot also use an ORDER BY clause to sort the results. In other words, ROLLUP and ORDER BY are mutually exclusive. However, you still have some control over sort order. GROUP BY in MySQL sorts results, and you can use explicit ASC and DESC keywords with columns named in the GROUP BY list to specify sort order for individual columns. (The higher-level summary rows added by ROLLUP still appear after the rows from which they are calculated, regardless of the sort order.)

LIMIT can be used to restrict the number of rows returned to the client. LIMIT is applied after ROLLUP, so the limit applies against the extra rows added by ROLLUP. For example:

```
mysql> SELECT year, country, product, SUM(profit)
    -> FROM sales
    -> GROUP BY year, country, product WITH ROLLUP
    -> LIMIT 5;
+------+---------+------------+-------------+
| year | country | product    | SUM(profit) |
+------+---------+------------+-------------+
| 2000 | Finland | Computer   |        1500 |
| 2000 | Finland | Phone      |         100 |
| 2000 | Finland | NULL       |        1600 |
| 2000 | India   | Calculator |         150 |
| 2000 | India   | Computer   |        1200 |
+------+---------+------------+-------------+
```

Using LIMIT with ROLLUP may produce results that are more difficult to interpret, because you have less context for understanding the super-aggregate rows.

The NULL indicators in each super-aggregate row are produced when the row is sent to the client. The server looks at the columns named in the GROUP BY clause following the leftmost one that has changed value. For any column in the result set with a name that is a lexical match to any of those names, its value is set to NULL. (If you specify grouping columns by column number, the server identifies which columns to set to NULL by number.)

Because the NULL values in the super-aggregate rows are placed into the result set at such a late stage in query processing, you cannot test them as NULL values within the query itself. For example, you cannot add HAVING product IS NULL to the query to eliminate from the output all but the super-aggregate rows.

On the other hand, the NULL values do appear as NULL on the client side and can be tested as such using any MySQL client programming interface.

## 5.9.3 GROUP BY with Hidden Fields

MySQL extends the use of GROUP BY so that you can use columns or calculations in the SELECT list that don't appear in the GROUP BY clause. 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 standard SQL, you would have to add customer.name to the GROUP BY clause. In MySQL, the name is redundant if you don't run in ANSI mode.

Do *not* use this feature if the columns you omit from the GROUP BY part are not unique in the group! You will get unpredictable results.

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(RPAD(sort,6,' '),column)),7)
```

Note that if you are using MySQL 3.22 (or earlier) or if you are trying to follow standard SQL, you can't 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;
```

In MySQL 3.23 and up, aliases are unnecessary. You can use expressions in GROUP BY and ORDER BY clauses. For example:

```
mysql> SELECT id, FLOOR(value/100) FROM tbl_name ORDER BY RAND();
```

# 6 SQL Statement Syntax

This chapter describes the syntax for the SQL statements supported in MySQL.

## 6.1 Data Manipulation Statements

### 6.1.1 DELETE Syntax

Single-table syntax:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name
       [WHERE where_definition]
       [ORDER BY ...]
       [LIMIT row_count]
```

Multiple-table syntax:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
       tbl_name[.*] [, tbl_name[.*] ...]
       FROM table_references
       [WHERE where_definition]
```

Or:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
       FROM tbl_name[.*] [, tbl_name[.*] ...]
       USING table_references
       [WHERE where_definition]
```

`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` statement with no `WHERE` clause, all rows are deleted. A faster way to do this, when you don't want to know the number of deleted rows, is to use `TRUNCATE` table. See Section 6.1.9, "`TRUNCATE` Syntax."

In MySQL 3.23, `DELETE` without a `WHERE` clause returns zero as the number of affected records.

In MySQL 3.2.3, 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 that includes a `WHERE` clause with an expression that is true for every row. For example:

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

This is much slower than `DELETE FROM` *`tbl_name`* with no `WHERE` clause, because it deletes rows one at a time.

If you delete the row containing the maximum value for an `AUTO_INCREMENT` column, the value will be reused for an `ISAM` or `BDB` table, but not for a `MyISAM` or `InnoDB` table. If you delete all rows in the table with `DELETE FROM` *`tbl_name`* (without a `WHERE`) in `AUTOCOMMIT` mode, the sequence starts over for all table types except for `InnoDB` and (as of MySQL 4.0) `MyISAM`. There are some exceptions to this behavior for `InnoDB` tables, discussed in the `InnoDB` chapter of the *MySQL Administrator's Guide*.

For `MyISAM` and `BDB` tables, you can specify an `AUTO_INCREMENT` secondary column in a multiple-column key. In this case, reuse of values deleted from the top of the sequence occurs even for `MyISAM` tables.

The `DELETE` statement supports the following modifiers:

- If you specify the `LOW_PRIORITY` keyword, execution of the `DELETE` is delayed until no other clients are reading from the table.
- For `MyISAM` tables, if you specify the `QUICK` keyword, the storage engine does not merge index leaves during delete, which may speed up certain kinds of deletes.
- The `IGNORE` keyword causes MySQL to ignore all errors during the process of deleting rows. (Errors encountered during the parsing stage are processed in the usual manner.) Errors that are ignored due to the use of this option are returned as warnings. This option first appeared in MySQL 4.1.1.

The speed of delete operations may also be affected by factors discussed in the optimization chapter of the *MySQL Administrator's Guide*.

In `MyISAM` tables, 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 `myisamchk` utility to reorganize tables. `OPTIMIZE TABLE` is easier, but `myisamchk` is faster. See Section 6.5.2.5, "`OPTIMIZE TABLE` Syntax."

The MySQL-specific `LIMIT` *`row_count`* 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` statement doesn't take too much time. You can simply repeat the `DELETE` statement until the number of affected rows is less than the `LIMIT` value.

If the DELETE statement includes an ORDER BY clause, the rows are deleted in the order specified by the clause. This is really useful only in conjunction with LIMIT. For example, the following statement finds rows matching the WHERE clause, sorts them in timestamp order, and deletes the first (oldest) one:

```
DELETE FROM somelog
WHERE user = 'jcole'
ORDER BY timestamp
LIMIT 1
```

ORDER BY can be used with DELETE beginning with MySQL 4.0.0.

From MySQL 4.0, you can specify multiple tables in the DELETE statement to delete rows from one or more tables depending on a particular condition in multiple tables. However, you cannot use ORDER BY or LIMIT in a multiple-table DELETE.

The first multiple-table DELETE syntax is supported starting from MySQL 4.0.0. The second is supported starting from MySQL 4.0.2. The *table_references* part lists the tables involved in the join. Its syntax is described in Section 6.1.7.1, "JOIN Syntax."

For the first syntax, only matching rows from the tables listed before the FROM clause are deleted. For the second syntax, only matching rows from the tables listed in the FROM clause (before the USING clause) are deleted. The effect is that you can delete rows from many tables at the same time and also have additional tables that are used for searching:

```
DELETE t1,t2 FROM t1,t2,t3 WHERE t1.id=t2.id AND t2.id=t3.id;
```

Or:

```
DELETE FROM t1,t2 USING t1,t2,t3 WHERE t1.id=t2.id AND t2.id=t3.id;
```

These statements use all three files when searching for rows to delete, but delete matching rows only from tables t1 and t2.

The examples show inner joins using the comma operator, but multiple-table DELETE statements can use any type of join allowed in SELECT statements, such as LEFT JOIN.

The syntax allows .* after the table names for compatibility with Access.

If you use a multiple-table DELETE statement involving InnoDB tables for which there are foreign key constraints, the MySQL optimizer might process tables in an order that differs from that of their parent/child relationship. In this case, the statement fails and rolls back. Instead, delete from a single table and rely on the ON DELETE capabilities that InnoDB provides to cause the other tables to be modified accordingly.

**Note:** In MySQL 4.0, you should refer to the table names to be deleted with the true table name. In MySQL 4.1, you must use the alias (if one was given) when referring to a table name:

In MySQL 4.0:

```
DELETE test FROM test AS t1, test2 WHERE ...
```

In MySQL 4.1:

```
DELETE t1 FROM test AS t1, test2 WHERE ...
```

The reason we didn't make this change in 4.0 is that we didn't want to break any old 4.0 applications that were using the old syntax.

## 6.1.2 DO Syntax

```
DO expr [, expr] ...
```

`DO` executes the expressions but doesn't return any results. This is shorthand for `SELECT` *`expr`*`, ...`, but has the advantage that it's slightly faster when you don't care about the result.

`DO` is useful mainly with functions that have side effects, such as `RELEASE_LOCK()`.

## 6.1.3 HANDLER Syntax

```
HANDLER tbl_name OPEN [ AS alias ]
HANDLER tbl_name READ index_name { = | >= | <= | < } (value1,value2,...)
    [ WHERE where_condition ] [LIMIT ... ]
HANDLER tbl_name READ index_name { FIRST | NEXT | PREV | LAST }
    [ WHERE where_condition ] [LIMIT ... ]
HANDLER tbl_name READ { FIRST | NEXT }
    [ WHERE where_condition ] [LIMIT ... ]
HANDLER tbl_name CLOSE
```

The `HANDLER` statement provides direct access to table storage engine interfaces. It is available for `MyISAM` tables as MySQL 4.0.0 and `InnoDB` tables as of MySQL 4.0.3.

The `HANDLER ... OPEN` statement opens a table, making it accessible via subsequent `HANDLER ... READ` statements. This table object is not shared by other threads and is not closed until the thread calls `HANDLER ... CLOSE` or the thread terminates. If you open the table using an alias, further references to the table with other `HANDLER` statements must use the alias rather than the table name.

The first `HANDLER ... READ` syntax fetches a row where the index specified satisfies the given values and the `WHERE` condition is met. If you have a multiple-column index, specify the index column values as a comma-separated list. Either specify values for all the columns in the index, or specify values for a leftmost prefix of the index columns. Suppose that an index includes three columns named `col_a`, `col_b`, and `col_c`, in that order. The `HANDLER` statement can specify values for all three columns in the index, or for the columns in a leftmost prefix. For example:

```
HANDLER ... index_name = (col_a_val,col_b_val,col_c_val) ...
HANDLER ... index_name = (col_a_val,col_b_val) ...
HANDLER ... index_name = (col_a_val) ...
```

The second `HANDLER ... READ` syntax fetches a row from the table in index order that matches the `WHERE` condition.

The third `HANDLER ... READ` syntax fetches a row from the table in natural row order that matches the `WHERE` condition. It is faster than `HANDLER` *`tbl_name`* `READ` *`index_name`* when a full table scan is desired. Natural row order is the order in which rows are stored in a `MyISAM` table data file. This statement works for `InnoDB` tables as well, but there is no such concept because there is no separate data file.

Without a `LIMIT` clause, all forms of `HANDLER ... READ` fetch a single row if one is available. To return a specific number of rows, include a `LIMIT` clause. It has the same syntax as for the `SELECT` statement. See Section 6.1.7, "`SELECT` Syntax."

`HANDLER ... CLOSE` closes a table that was opened with `HANDLER ... OPEN`.

**Note:** To use the `HANDLER` interface to refer to a table's `PRIMARY KEY`, use the quoted identifier `` `PRIMARY` ``:

```
HANDLER tbl_name READ `PRIMARY` > (...);
```

`HANDLER` is a somewhat low-level statement. For example, it does not provide consistency. That is, `HANDLER ... OPEN` does *not* take a snapshot of the table, and does *not* lock the table. This means that after a `HANDLER ... OPEN` statement is issued, table data can be modified (by this or any other thread) and these modifications might appear only partially in `HANDLER ... NEXT` or `HANDLER ... PREV` scans.

There are several reasons to use the `HANDLER` interface instead of normal `SELECT` statements:

- `HANDLER` is faster than `SELECT`:
  - A designated storage engine handler object is allocated for the `HANDLER ... OPEN`. The object is reused for the following `HANDLER` statements for the table; it need not be reinitialized for each one.
  - There is less parsing involved.
  - There is no optimizer or query-checking overhead.
  - The table doesn't have to be locked between two handler requests.
  - The handler interface doesn't have to provide a consistent look of the data (for example, dirty reads are allowed), so the storage engine can use optimizations that `SELECT` doesn't normally allow.
- `HANDLER` makes it much easier to port applications that use an `ISAM`-like interface to MySQL.
- `HANDLER` allows you to traverse a database in a manner that is not easy (or perhaps even impossible) to do with `SELECT`. The `HANDLER` interface is a more natural way to look at data when working with applications that provide an interactive user interface to the database.

## 6.1.4 INSERT Syntax

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
    [INTO] tbl_name [(col_name,...)]
    VALUES ({expr | DEFAULT},...),(...),...
    [ ON DUPLICATE KEY UPDATE col_name=expr, ... ]
```

Or:

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
    [INTO] tbl_name
    SET col_name={expr | DEFAULT}, ...
    [ ON DUPLICATE KEY UPDATE col_name=expr, ... ]
```

Or:

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

`INSERT` inserts new rows into an existing table. The `INSERT ... VALUES` and `INSERT ... SET` forms of the statement insert 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 `INSERT ... SET` syntax is supported in MySQL 3.22.10 or later. `INSERT ... SELECT` is discussed further in See Section 6.1.4.1, "`INSERT ... SELECT` Syntax."

*`tbl_name`* is the table into which rows should be inserted. The columns for which the statement provides values can be specified as follows:

- The column name list or the `SET` clause indicates the columns explicitly.
- If you do not specify the column list for `INSERT ... VALUES` or `INSERT ... SELECT`, values for every column in the table 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.

Column values can be given in several ways:

- 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 6.2.5, "`CREATE TABLE` Syntax."

  MySQL always has a default value for all columns. This is something that is imposed on MySQL to be able to work with both transactional and non-transactional tables.

  Our view is that column content checking should be done in the application and not in the database server.

Note: If you want INSERT statements to generate an error unless you explicitly specify values for all columns that require a non-NULL value, you can configure MySQL using the DONT_USE_DEFAULT_FIELDS compile option. This behavior is available only if you compile MySQL from source.

- You can use the keyword DEFAULT to explicitly set a column to its default value. (New in MySQL 4.0.3.) This makes it easier to write INSERT statements that assign values to all but a few columns, because it allows you to avoid writing an incomplete VALUES list that does not include a value for each column in the table. Otherwise, you would have to write out the list of column names corresponding to each value in the VALUES list.
- If both the column list and the VALUES list are empty, INSERT creates a row with each column set to its default value.

```
mysql> INSERT INTO tbl_name () VALUES();
```

- An expression *expr* can refer to any column that was set earlier in a value list. For example, you can do this because the value for col2 refers to col1, which has already been assigned:

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

But you cannot do this because the value for col1 refers to col2, which is assigned after col1:

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

The INSERT statement supports the following modifiers:

- If you specify the DELAYED keyword, the server puts the row or rows to be inserted into a buffer, and the client issuing the INSERT DELAYED statement then can continue on. If the table is busy, the server holds the rows. When the table becomes free, it begins inserting rows, checking periodically to see whether there are new read requests for the table. If there are, the delayed row queue is suspended until the table becomes free again. See Section 6.1.4.2, "INSERT DELAYED Syntax."
- If you specify the LOW_PRIORITY keyword, execution of the INSERT is delayed until no other clients are reading from the table. This includes other clients that began reading while existing clients are reading, and while the INSERT LOW_PRIORITY statement is waiting. It is possible, therefore, for a client that issues an INSERT LOW_PRIORITY statement to wait for a very long time (or even forever) in a read-heavy environment. (This is in contrast to INSERT DELAYED, which lets the client continue at once.) See Section 6.1.4.2, "INSERT DELAYED Syntax." Note that LOW_PRIORITY should normally not be used with MyISAM tables because doing so disables concurrent inserts.
- If you specify the IGNORE keyword in an INSERT with many rows, any rows that duplicate an existing UNIQUE index or PRIMARY KEY value 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 determine with the mysql_info() C API function how many rows were inserted into the table.

If you specify the `ON DUPLICATE KEY UPDATE` clause (new in MySQL 4.1.0), and a row is inserted that would cause a duplicate value in a `UNIQUE` index or `PRIMARY KEY`, an `UPDATE` of the old row is performed. For example, if column `a` is declared as `UNIQUE` and already contains the value `1`, the following two statements have identical effect:

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3)
    -> ON DUPLICATE KEY UPDATE c=c+1;

mysql> UPDATE table SET c=c+1 WHERE a=1;
```

**Note:** If column `b` is unique too, the `INSERT` would be equivalent to this `UPDATE` statement instead:

```
mysql> UPDATE table SET c=c+1 WHERE a=1 OR b=2 LIMIT 1;
```

If `a=1 OR b=2` matches several rows, only *one* row is updated! In general, you should try to avoid using the `ON DUPLICATE KEY` clause on tables with multiple `UNIQUE` keys.

As of MySQL 4.1.1, you can use the `VALUES(`*`col_name`*`)` function in the `UPDATE` clause to refer to column values from the `INSERT` part of the `INSERT ... UPDATE` statement. In other words, `VALUES(`*`col_name`*`)` in the `UPDATE` clause refers to the value of *`col_name`* that would be inserted if no duplicate-key conflict occurred. This function is especially useful in multiple-row inserts. The `VALUES()` function is meaningful only in `INSERT ... UPDATE` statements and returns `NULL` otherwise.

Example:

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3),(4,5,6)
    -> ON DUPLICATE KEY UPDATE c=VALUES(a)+VALUES(b);
```

That statement is identical to the following two statements:

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3)
    -> ON DUPLICATE KEY UPDATE c=3;
mysql> INSERT INTO table (a,b,c) VALUES (4,5,6)
    -> ON DUPLICATE KEY UPDATE c=9;
```

When you use `ON DUPLICATE KEY UPDATE`, the `DELAYED` option is ignored.

You can find the value used for an `AUTO_INCREMENT` column by using the `LAST_INSERT_ID()` function. From within the C API, use the `mysql_insert_id()` function. However, note that the two functions do not behave quite identically under all circumstances. The behavior of `INSERT` statements with respect to `AUTO_INCREMENT` columns is discussed further in Section 5.8.3, "Information Functions."

If you use an `INSERT ... VALUES` statement with multiple value lists or `INSERT ... SELECT`, the statement returns an information string in this format:

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

`Records` indicates the number of rows processed by the statement. (This is not necessarily the number of rows actually inserted. `Duplicates` can be non-zero.) `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`. For multiple-row `INSERT` statements or `INSERT ... SELECT` statements, the column is set to the default value appropriate for the column type. This is `0` for numeric types, the empty string (`''`) for string types, and the "zero" value for date and time types.
- Setting a numeric column to a value that lies outside the column's range. The value is clipped to the closest endpoint of the range.
- Assigning a value such as `'10.34 a'` to a numeric column. The trailing non-numeric text is stripped off and the remaining numeric part is inserted. If the string value has no leading numeric part, the column is set to `0`.
- Inserting a string into a string column (`CHAR`, `VARCHAR`, `TEXT`, or `BLOB`) 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.

If you are using the C API, the information string can be obtained by invoking the `mysql_info()` function.

### 6.1.4.1 `INSERT ... SELECT` Syntax

```
INSERT [LOW_PRIORITY] [IGNORE] [INTO] tbl_name [(column_list)]
    SELECT ...
```

With `INSERT ... SELECT`, you can quickly insert many rows into a table from one or many tables.

For example:

```
INSERT INTO tbl_temp2 (fld_id)
    SELECT tbl_temp1.fld_order_id
    FROM tbl_temp1 WHERE tbl_temp1.fld_order_id > 100;
```

The following conditions hold for an `INSERT ... SELECT` statement:

- Prior to MySQL 4.0.1, `INSERT ... SELECT` implicitly operates in `IGNORE` mode. As of MySQL 4.0.1, specify `IGNORE` explicitly to ignore records that would cause duplicate-key violations.
- Do not use `DELAYED` with `INSERT ... SELECT`.
- Prior to MySQL 4.0.14, the target table of the `INSERT` statement cannot appear in the `FROM` clause of the `SELECT` part of the query. This limitation is lifted in 4.0.14.

- AUTO_INCREMENT columns work as usual.
- To ensure that the binary log can be used to re-create the original tables, MySQL will not allow concurrent inserts during INSERT ... SELECT.

You can use REPLACE instead of INSERT to overwrite old rows. REPLACE is the counterpart to INSERT IGNORE in the treatment of new rows that contain unique key values that duplicate old rows: The new rows are used to replace the old rows rather than being discarded.

### 6.1.4.2 INSERT DELAYED Syntax

```
INSERT DELAYED ...
```

The DELAYED option for the INSERT statement is a MySQL extension to standard SQL that is very useful if you have clients that can't wait for the INSERT to complete. This is a common problem when you use MySQL for logging and you also periodically run SELECT and UPDATE statements that take a long time to complete. DELAYED was introduced in MySQL 3.22.15.

When a client uses INSERT DELAYED, it gets an okay from the server at once, and the row is queued to be inserted when the table is not in use by any other thread.

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.

There are some constraints on the use of DELAYED:

- INSERT DELAYED works only with MyISAM and ISAM tables. For MyISAM tables, if there are no free blocks in the middle of the data file, concurrent SELECT and INSERT statements are supported. Under these circumstances, you very seldom need to use INSERT DELAYED with MyISAM.
- INSERT DELAYED should be used only for INSERT statements that specify value lists. This is enforced as of MySQL 4.0.18. The server ignores DELAYED for INSERT DELAYED ... SELECT statements.
- The server ignores DELAYED for INSERT DELAYED ... ON DUPLICATE UPDATE statements.
- Because the statement returns immediately before the rows are inserted, you cannot use LAST_INSERT_ID() to get the AUTO_INCREMENT value the statement might generate.
- DELAYED rows are not visible to SELECT statements until they actually have been inserted.

Note that currently the queued rows are held only in memory until they are inserted into the table. This means that if you terminate mysqld forceably (for example, with kill -9) or if mysqld dies unexpectedly, any queued rows that have not been written to disk are lost!

The following describes in detail what happens when you use the DELAYED option to INSERT or REPLACE. In this description, the "thread" is the thread that received an INSERT DELAYED statement 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 process all DELAYED statements for the table, if no such handler already exists.
- The thread checks whether 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 to the client program.
- The client cannot obtain from the server the number of duplicate records or the AUTO_INCREMENT value for the resulting row, 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 binary log is updated by the handler thread when the row is inserted into the table. In case of multiple-row inserts, the binary log is updated when the first row is inserted.
- After every delayed_insert_limit rows are written, the handler checks whether 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 statements 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 requesting INSERT DELAYED waits until there is room in the queue. This is done to ensure that the mysqld server doesn't use all memory for the delayed memory queue.
- The handler thread shows up in the MySQL process list with delayed_insert in the Command column. It will be killed if you execute a FLUSH TABLES statement or kill it with KILL *thread_id*. However, before exiting, it will first store all queued rows into the table. During this time it will not accept any new INSERT statements from another thread. If you execute an INSERT DELAYED statement after this, a new handler thread will be created.

  Note that this means that INSERT DELAYED statements have higher priority than normal INSERT statements if there is an INSERT DELAYED handler already running! Other update statements will have to wait until the INSERT DELAYED queue is empty, someone terminates the handler thread (with KILL *thread_id*), or someone executes FLUSH TABLES.
- The following status variables provide information about INSERT DELAYED statements:

  | Status Variable | Meaning |
  |---|---|
  | Delayed_insert_threads | Number of handler threads |
  | Delayed_writes | Number of rows written with INSERT DELAYED |
  | Not_flushed_delayed_rows | 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.

Note that `INSERT DELAYED` is slower than a normal `INSERT` if the table is not in use. There is also the additional overhead for the server to handle a separate thread for each table for which there are delayed rows. This means that you should use `INSERT DELAYED` only when you are really sure that you need it!

## 6.1.5 LOAD DATA INFILE Syntax

```
LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE 'file_name.txt'
    [REPLACE | IGNORE]
    INTO TABLE tbl_name
    [FIELDS
        [TERMINATED BY '\t']
        [[OPTIONALLY] ENCLOSED BY '']
        [ESCAPED BY '\\' ]
    ]
    [LINES
        [STARTING BY '']
        [TERMINATED BY '\n']
    ]
    [IGNORE number LINES]
    [(col_name,...)]
```

The `LOAD DATA INFILE` statement reads rows from a text file into a table at a very high speed.

You can also load data files by using the `mysqlimport` utility; it operates by sending a `LOAD DATA INFILE` statement 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.

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

If you specify the `CONCURRENT` keyword with a `MyISAM` table that satisfies the condition for concurrent inserts (that is, it contains no free blocks in the middle), then other threads can retrieve data from the table while `LOAD DATA` is executing. Using this option affects the performance of `LOAD DATA` a bit, even if no other thread is using the table at the same time.

If the `LOCAL` keyword is specified, it is interpreted with respect to the client end of the connection:

- If `LOCAL` is specified, the file is read by the client program on the client host and sent to the server.
- If `LOCAL` is not specified, the file must be located on the server host and is read directly by 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.

Using LOCAL is a bit slower than letting the server access the files directly, because the contents of the file must be sent over the connection by the client to the server. On the other hand, you do not need the FILE privilege to load local files.

As of MySQL 3.23.49 and MySQL 4.0.2 (4.0.13 on Windows), LOCAL works only if your server and your client both have been enabled to allow it. For example, if mysqld was started with --local-infile=0, LOCAL will not work.

If you need LOAD DATA to read from a pipe, you can use the following technique:

```
mkfifo /mysql/db/x/x
chmod 666 /mysql/db/x/x
cat < /dev/tcp/10.1.1.12/4711 > /mysql/db/x/x
mysql -e "LOAD DATA INFILE 'x' INTO TABLE x" x
```

If you are using a version of MySQL older than 3.23.25, you can use this technique only with LOAD DATA LOCAL INFILE.

If you are using MySQL before Version 3.23.24, you can't read from a FIFO with LOAD DATA INFILE. If you need to read from a FIFO (for example, the output from gunzip), use LOAD DATA LOCAL INFILE instead.

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 default database.

Note that these rules mean that a file named as ./myfile.txt is read from the server's data directory, whereas the same file named as myfile.txt is read from the database directory of the default database. For example, the following LOAD DATA statement reads the file data.txt from the database directory for db1 because db1 is the current database, even though the statement explicitly loads the file into a table in the db2 database:

```
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`, input rows replace existing rows (in other words, rows that have the same value for a primary or unique index as an existing row). See Section 6.1.6, "`REPLACE` Syntax."

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, the behavior depends on whether or not the `LOCAL` keyword is specified. Without `LOCAL`, an error occurs when a duplicate key value is found, and the rest of the text file is ignored. With `LOCAL`, the default behavior is the same as if `IGNORE` is specified; this is because the server has no way to stop transmission of the file in the middle of the operation.

If you want to ignore foreign key constraints during the load operation, you can issue a `SET FOREIGN_KEY_CHECKS=0` statement before executing `LOAD DATA`.

If you use `LOAD DATA INFILE` on an empty `MyISAM` table, all non-unique indexes are created in a separate batch (as for `REPAIR TABLE`). This normally makes `LOAD DATA INFILE` much faster when you have many indexes. Normally this is very fast, but in some extreme cases, you can create the indexes even faster by turning them off with `ALTER TABLE .. DISABLE KEYS` before loading the file into the table and using `ALTER TABLE .. ENABLE KEYS` to re-create the indexes after loading the file.

`LOAD DATA INFILE` is the complement of `SELECT ... INTO OUTFILE`. See Section 6.1.7, "`SELECT` Syntax." To write data from a table to a file, use `SELECT ... INTO OUTFILE`. To read the file back into a table, use `LOAD DATA INFILE`. The syntax of the `FIELDS` and `LINES` clauses is the same for both statements. 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' STARTING BY ''
```

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

- Look for line boundaries at newlines.
- Do not skip over any line prefix.
- 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.

**Note:** If you have generated the text file on a Windows system, you might have to use `LINES TERMINATED BY '\r\n'` to read the file properly, because Windows programs typically use two characters as a line terminator. Some programs, such as `WordPad`, might use `\r` as a line terminator when writing files. To read such files, use `LINES TERMINATED BY '\r'`.

If all the lines you want to read in have a common prefix that you want to ignore, you can use `LINES STARTING BY '`*`prefix_string`*`'` to skip over the prefix. If a line doesn't include the prefix, the entire line is skipped.

The `IGNORE` *`number`* `LINES` option can be used to ignore lines at the start of the file. For example, you can use `IGNORE 1 LINES` to skip over an initial header line containing column names:

```
mysql> LOAD DATA INFILE '/tmp/test.txt'
    -> 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 statements must match. Otherwise, `LOAD DATA INFILE` will not interpret the contents of the file properly. Suppose that you use `SELECT ... INTO OUTFILE` to write a file with fields delimited by commas:

```
mysql> SELECT * INTO OUTFILE 'data.txt'
    ->          FIELDS TERMINATED BY ','
    ->          FROM table2;
```

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 here, 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 within double quotes. If lines in the file are terminated by newlines, the statement shown here 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 can 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, LINES STARTING BY, and LINES TERMINATED BY values can 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.

To read a file containing jokes that are separated by lines consisting of %%, you can do this

```
mysql> CREATE TABLE jokes
    ->     (a INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->     joke TEXT NOT NULL);
mysql> LOAD DATA INFILE '/tmp/jokes.txt' INTO TABLE jokes
    ->     FIELDS TERMINATED BY ''
    ->     LINES TERMINATED BY '\n%%\n' (joke);
```

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 here:

```
"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 preceding output just shown would appear as follows 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.

If the field begins with the `ENCLOSED BY` character, instances of that character are recognized as terminating a field value only if followed by the field or line `TERMINATED BY` sequence. To avoid ambiguity, occurrences of the `ENCLOSED BY` character within a field value can be doubled and will be interpreted as a single instance of the character. For example, if `ENCLOSED BY '"'` is specified, quotes are handled as shown here:

```
"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 and `NULL` is output as `NULL`, not `\N`. 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`' (for example, `\0` or `\N` if the escape character is '\'). These sequences are interpreted as ASCII NUL (a zero-valued byte) and `NULL`. The rules for `NULL` handling are described later in this section.

For more information about '\'-escape syntax, see Section 2.1, "Literal Values."

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 (but you can still have a line terminator). 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 seven-character fields. On input, values for the column are obtained by reading seven characters.

  `LINES TERMINATED BY` is still used to separate lines. If a line doesn't contain all fields, the rest of the columns are set to their default values. If you don't have a line terminator, you should set this to `''`. In this case, the text file must contain all fields for each row.

  Fixed-row format also affects handling of `NULL` values, as described later. Note that fixed-size format will not work if you are using a multi-byte character set.

Handling of `NULL` values varies according to the `FIELDS` and `LINES` options in use:

- For the default `FIELDS` and `LINES` values, `NULL` is written as a field value of `\N` for output, and a field value of `\N` is read as `NULL` for input (assuming that 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 because 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;
```

By default, when no column list is provided at the end of the `LOAD DATA INFILE` statement, input lines are expected to contain a field for each table column. If you want to load only some of a table's columns, specify a column list:

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

You must also specify a column 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 an input line has too many fields, the extra fields are ignored and the number of warnings is incremented.

If an input line has too few fields, the table columns for which input fields are missing are set to their default values. Default value assignment is described in Section 6.2.5, "`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 4.3, "Date and Time Types."

These are the same values that result if you assign an empty string explicitly to a string, numeric, or date or time type explicitly in an `INSERT` or `UPDATE` statement.

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

`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!

When the `LOAD DATA INFILE` statement finishes, it returns an information string in the following format:

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

If you are using the C API, you can get information about the statement by calling the `mysql_info()` C API function.

Warnings occur under the same circumstances as when values are inserted via the `INSERT` statement (see Section 6.1.4, "`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 be used only as an indication of whether everything went well.

From MySQL 4.1.1 on, you can use `SHOW WARNINGS` to get a list of the first `max_error_count` warnings as information about what went wrong. See Section 6.5.3.20, "`SHOW WARNINGS` Syntax."

Before MySQL 4.1.1, only a warning count is available to indicate that something went wrong. If you get warnings and want to know exactly why you got them, one way to do this is to dump the table into another file using `SELECT ... INTO OUTFILE` and compare the file to your original input file.

## 6.1.6 REPLACE Syntax

```
REPLACE [LOW_PRIORITY | DELAYED]
    [INTO] tbl_name [(col_name,...)]
    VALUES ({expr | DEFAULT},...),(...),...
```

Or:

```
REPLACE [LOW_PRIORITY | DELAYED]
    [INTO] tbl_name
    SET col_name={expr | DEFAULT}, ...
```

Or:

```
REPLACE [LOW_PRIORITY | DELAYED]
    [INTO] tbl_name [(col_name,...)]
    SELECT ...
```

`REPLACE` works exactly like `INSERT`, except that if an old record in the table has the same value as a new record for a `PRIMARY KEY` or a `UNIQUE` index, the old record is deleted before the new record is inserted. See Section 6.1.4, "`INSERT` Syntax."

Note that unless the table has a `PRIMARY KEY` or `UNIQUE` index, using a `REPLACE` statement makes no sense. It becomes equivalent to `INSERT`, because there is no index to be used to determine whether a new row duplicates another.

Values for all columns are taken from the values specified in the `REPLACE` statement. Any missing columns are set to their default values, just as happens for `INSERT`. You can't refer to values from the old row and use them in the new row. It appeared that you could do this in some old MySQL versions, but that was a bug that has been corrected.

To be able to use `REPLACE`, you must have `INSERT` and `DELETE` privileges for the table.

The `REPLACE` statement returns a count to indicate the number of rows affected. This is the sum of the rows deleted and inserted. If the count is 1 for a single-row `REPLACE`, a row was inserted and no rows were deleted. If the count is greater than 1, one or more old rows were deleted before the new row was inserted. It is possible for a single row to replace more than one old row if the table contains multiple unique indexes and the new row duplicates values for different old rows in different unique indexes.

The affected-rows count makes it easy to determine whether `REPLACE` only added a row or whether it also replaced any rows: Check whether the count is 1 (added) or greater (replaced).

If you are using the C API, the affected-rows count can be obtained using the `mysql_affected_rows()` function.

Here follows in more detail the algorithm that is used (it is also used with `LOAD DATA ... REPLACE`):

1. Try to insert the new row into the table
2. While the insertion fails because a duplicate-key error occurs for a primary or unique key:
   a. Delete from the table the conflicting row that has the duplicate key value
   b. Try again to insert the new row into the table

## 6.1.7 SELECT Syntax

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
      [HIGH_PRIORITY]
      [STRAIGHT_JOIN]
      [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
      [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr,...
    [INTO OUTFILE 'file_name' export_options
      | INTO DUMPFILE 'file_name']
    [FROM table_references
      [WHERE where_definition]
      [GROUP BY {col_name | expr | position}
        [ASC | DESC], ... [WITH ROLLUP]]
      [HAVING where_definition]
      [ORDER BY {col_name | expr | position}
        [ASC | DESC] ,...]
      [LIMIT [offset,{] row_count | row_count OFFSET offset}]
      [PROCEDURE procedure_name(argument_list)]
      [FOR UPDATE | LOCK IN SHARE MODE]]
```

`SELECT` is used to retrieve rows selected from one or more tables. Support for `UNION` statements and subqueries is available as of MySQL 4.0 and 4.1, respectively. See Section 6.1.7.2, "`UNION` Syntax," and Section 6.1.8, "Subquery Syntax."

- Each *`select_expr`* indicates a column you want to retrieve.
- *`table_references`* indicates the table or tables from which to retrieve rows. Its syntax is described in Section 6.1.7.1, "`JOIN` Syntax."
- *`where_definition`* indicates any conditions that selected rows must satisfy.

`SELECT` can also be used to retrieve rows computed without reference to any table.

For example:

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

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

- A *`select_expr`* can be given an alias using `AS` *`alias_name`*. The alias is used as the expression's column name and can be used in `GROUP BY`, `ORDER BY`, or `HAVING` clauses. For example:

  ```
  mysql> SELECT CONCAT(last_name,', ',first_name) AS full_name
      -> FROM mytable ORDER BY full_name;
  ```

  The `AS` keyword is optional when aliasing a *`select_expr`*. The preceding example could have been written like this:

  ```
  mysql> SELECT CONCAT(last_name,', ',first_name) full_name
      -> FROM mytable ORDER BY full_name;
  ```

  Because the `AS` is optional, a subtle problem can occur if you forget the comma between two `SELECT` expressions: MySQL interprets the second as an alias name. For example, in the following statement, `columnb` is treated as an alias name:

  ```
  mysql> SELECT columna columnb FROM mytable;
  ```

- It is not allowable to use a column alias in a `WHERE` clause, because the column value might not yet be determined when the `WHERE` clause is executed. See Section A.1.4, "Problems with Column Aliases."
- 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 6.1.7.1, "`JOIN` Syntax." For each table specified, you can optionally specify an alias.

  ```
  tbl_name [[AS] alias]
      [[USE INDEX (key_list)]
        | [IGNORE INDEX (key_list)]
        | [FORCE INDEX (key_list)]]
  ```

The use of `USE INDEX`, `IGNORE INDEX`, `FORCE INDEX` to give the optimizer hints about how to choose indexes is described in Section 6.1.7.1, "`JOIN` Syntax."

As of MySQL 4.0.14, you can use `SET max_seeks_for_key=`*`value`* as an alternative way to force MySQL to prefer key scans instead of table scans.

- You can refer to a table within the current database as *`tbl_name`* (within the current database), or as *`db_name.tbl_name`* to explicitly specify a database. 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 unless the reference would be ambiguous. See Section 2.2, "Database, Table, Index, Column, and Alias Names," for examples of ambiguity that require the more explicit column reference forms.
- From MySQL 4.1.0 on, you are allowed to specify `DUAL` as a dummy table name in situations where no tables are referenced:

```
mysql> SELECT 1 + 1 FROM DUAL;
        -> 2
```

  `DUAL` is purely a compatibility feature. Some other servers require this syntax.
- A table reference can 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;
```

- In the `WHERE` clause, you can use any of the functions that MySQL supports, except for aggregate (summary) functions. See Chapter 5, "Functions and Operators."
- Columns selected for output can be referred to in `ORDER BY` and `GROUP BY` clauses using column names, column aliases, or column positions. Column positions are integers and 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 can be specified explicitly using the `ASC` keyword.

  Use of column positions is deprecated because the syntax has been removed from the SQL standard.
- If you use `GROUP BY`, output rows are sorted according to the `GROUP BY` columns as if you had an `ORDER BY` for the same columns. MySQL has extended the `GROUP BY` clause as of version 3.23.34 so that you can also specify `ASC` and `DESC` after columns named in the clause:

```
SELECT a, COUNT(b) FROM test_table GROUP BY a DESC
```

- MySQL extends the use of GROUP BY to allow you to select fields that are not mentioned in the GROUP BY clause. If you are not getting the results you expect from your query, please read the GROUP BY description. See Section 5.9, "Functions and Modifiers for Use with GROUP BY Clauses."
- As of MySQL 4.1.1, GROUP BY allows a WITH ROLLUP modifier. See Section 5.9.2, "GROUP BY Modifiers."
- The HAVING clause can refer to any column or alias named in a *select_expr*. It is applied nearly last, just before items are sent to the client, with no optimization. (LIMIT is applied after HAVING.)
- 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;
  ```

- The HAVING clause can refer to aggregate functions, which the WHERE clause cannot:

  ```
  mysql> SELECT user, MAX(salary) FROM users
      ->     GROUP BY user HAVING MAX(salary)>10;
  ```

  However, that does not work in older MySQL servers (before version 3.22.5). Instead, you can use a column alias in the select list and refer to the alias in the HAVING clause:

  ```
  mysql> SELECT user, MAX(salary) AS max_salary FROM users
      ->     GROUP BY user HAVING max_salary>10;
  ```

- The LIMIT clause can be used to constrain the number of rows returned by the SELECT statement. LIMIT takes one or two numeric arguments, which must be integer constants.

  With two arguments, the first argument specifies the offset of the first row to return, and 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
  ```

  For compatibility with PostgreSQL, MySQL also supports the LIMIT *row_count* OFFSET *offset* syntax.

  To retrieve all rows from a certain offset up to the end of the result set, you can use some large number for the second parameter. This statement retrieves all rows from the 96th row to the last:

  ```
  mysql> SELECT * FROM table LIMIT 95,18446744073709551615;
  ```

With one argument, the value specifies the number of rows to return from the beginning of the result set:

```
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'` syntax writes the selected rows to a file. The file is created on the server host, so you must have the `FILE` privilege to use this form of `SELECT`. The file cannot already exist, which among other things prevents files such as `/etc/passwd` and database tables from being destroyed.

  The `SELECT ... INTO OUTFILE` statement is intended primarily to let you very quickly dump a table on the server machine. If you want to create the resulting file on some client host other than the server host, you can't use `SELECT ... INTO OUTFILE`. In that case, you should instead use some command like `mysql -e "SELECT ..." > file_name` on the client host to generate the file.

  `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 6.1.5, "`LOAD DATA INFILE` Syntax."

  `FIELDS ESCAPED BY` controls how to write 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 and `NULL` is output as `NULL`, not `\N`. 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.

  The reason for the above is that you *must* escape any `FIELDS TERMINATED BY`, `ENCLOSED BY`, `ESCAPED BY`, or `LINES TERMINATED BY` characters to reliably be able to read the file back. ASCII NUL is escaped to make it easier to view with some pagers.

  The resulting file doesn't have to conform to SQL syntax, so nothing else need be escaped.

  Here is an example that produces a file in the comma-separated values format used by many programs:

```
SELECT a,b,a+b INTO OUTFILE '/tmp/result.text'
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
FROM test_table;
```

- If you use INTO DUMPFILE instead of INTO OUTFILE, MySQL writes only one row into the file, without any column or line termination and without performing any escape processing. This is useful if you want to store a BLOB value in a file.
- **Note:** Any file created by INTO OUTFILE or INTO DUMPFILE is writable by all users on the server host. The reason for this is that the MySQL server can't create a file that is owned by anyone other than the user it's running as (you should never run mysqld as root). The file thus must be world-writable so that you can manipulate its contents.
- A PROCEDURE clause names a procedure that should process the data in the result set.
- If you use FOR UPDATE on a storage engine that uses page or row locks, rows examined by the query are write-locked until the end of the current transaction.

Following the SELECT keyword, you can give a number of options that affect the operation of the statement.

The ALL, DISTINCT, and DISTINCTROW options specify whether duplicate rows should be returned. If none of these options are given, the default is ALL (all matching rows are returned). DISTINCT and DISTINCTROW are synonyms and specify that duplicate rows in the result set should be removed.

HIGH_PRIORITY, STRAIGHT_JOIN, and options beginning with SQL_ are MySQL extensions to standard SQL.

- HIGH_PRIORITY will give the SELECT higher priority than a statement that updates a table. You should use this only for queries that are very fast and must be done at once. A SELECT HIGH_PRIORITY query that is issued while the table is locked for reading will run even if there is already an update statement waiting for the table to be free.

  HIGH_PRIORITY cannot be used with SELECT statements that are part of a UNION.
- 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. STRAIGHT_JOIN also can be used in the *table_references* list. See Section 6.1.7.1, "JOIN Syntax."
- SQL_BIG_RESULT can be used with GROUP BY or DISTINCT to tell the optimizer that the result set will have many rows. In this case, MySQL will directly use disk-based temporary tables if needed. MySQL will also, in this case, prefer sorting to using a temporary table with a key on the GROUP BY elements.
- SQL_BUFFER_RESULT forces the result to be put into a temporary table. This helps MySQL free the table locks early and helps in cases where it takes a long time to send the result set to the client.
- 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 uses fast temporary tables to store the resulting table instead of using sorting. In MySQL 3.23 and up, this shouldn't normally be needed.

- `SQL_CALC_FOUND_ROWS` (available in MySQL 4.0.0 and up) tells MySQL to calculate how many rows there would be in the result set, disregarding any `LIMIT` clause. The number of rows can then be retrieved with `SELECT FOUND_ROWS()`. See Section 5.8.3, "Information Functions."

  Before MySQL 4.1.0, this option does not work with `LIMIT 0`, which is optimized to return instantly (resulting in a row count of 0).
- `SQL_CACHE` tells MySQL to store the query result in the query cache if you are using a `query_cache_type` value of `2` or `DEMAND`. For a query that uses `UNION` or subqueries, this option takes effect to be used in any `SELECT` of the query.
- `SQL_NO_CACHE` tells MySQL not to store the query result in the query cache. For a query that uses `UNION` or subqueries, this option takes effect to be used in any `SELECT` of the query.

### 6.1.7.1 `JOIN` Syntax

MySQL supports the following `JOIN` syntaxes for the *`table_references`* part of `SELECT` statements and multiple-table `DELETE` and `UPDATE` statements:

```
table_reference, table_reference
table_reference [INNER | CROSS] JOIN table_reference [join_condition]
table_reference STRAIGHT_JOIN table_reference
table_reference LEFT [OUTER] JOIN table_reference [join_condition]
table_reference NATURAL [LEFT [OUTER]] JOIN table_reference
{ OJ table_reference LEFT OUTER JOIN table_reference
    ON conditional_expr }
table_reference RIGHT [OUTER] JOIN table_reference [join_condition]
table_reference NATURAL [RIGHT [OUTER]] JOIN table_reference
```

*`table_reference`* is defined as:

```
tbl_name [[AS] alias]
    [[USE INDEX (key_list)]
      | [IGNORE INDEX (key_list)]
      | [FORCE INDEX (key_list)]]
```

*`join_condition`* is defined as:

```
ON conditional_expr | USING (column_list)
```

You should generally not have any conditions in the `ON` part that are used to restrict which rows you want in the result set, but rather specify these conditions in the `WHERE` clause. There are exceptions to this rule.

Note that `INNER JOIN` syntax allows a *`join_condition`* only from MySQL 3.23.17 on. The same is true for `JOIN` and `CROSS JOIN` only as of MySQL 4.0.11.

The `{ OJ ... LEFT OUTER JOIN ...}` syntax shown in the preceding list exists only for compatibility with ODBC.

- A table reference can 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;
mysql> SELECT t1.name, t2.salary FROM employee t1, info t2
    ->        WHERE t1.name = t2.name;
```

- The `ON` conditional is any conditional expression of the form that can be used in a `WHERE` clause.
- If there is no matching record for the right table in the `ON` or `USING` part 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` (that is, all rows in `table1` with no corresponding row in `table2`). This assumes that `table2.id` is declared `NOT NULL`.
- The `USING (`*`column_list`*`)` clause names a list of columns that must exist in both tables. The following two clauses are semantically identical:

```
a LEFT JOIN b USING (c1,c2,c3)
a LEFT JOIN b ON a.c1=b.c1 AND a.c2=b.c2 AND a.c3=b.c3
```

- The `NATURAL [LEFT] JOIN` of two tables is defined to be semantically equivalent to an `INNER JOIN` or a `LEFT JOIN` with a `USING` clause that names all columns that exist in both tables.
- `INNER JOIN` and `,` (comma) are semantically equivalent in the absence of a join condition: both will produce a Cartesian product between the specified tables (that is, each and every row in the first table will be joined onto all rows in the second table).
- `RIGHT JOIN` works analogously to `LEFT JOIN`. To keep code portable across databases, it's recommended to use `LEFT JOIN` instead of `RIGHT JOIN`.
- `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 for which the join optimizer puts the tables in the wrong order.

As of MySQL 3.23.12, you can give hints about which index MySQL should use when retrieving information from a table. By specifying `USE INDEX (`*`key_list`*`)`, you can tell MySQL to use only one of the possible indexes to find rows in the table. The alternative syntax `IGNORE INDEX (`*`key_list`*`)` can be used to tell MySQL to not use some particular index. These hints are useful if `EXPLAIN` shows that MySQL is using the wrong index from the list of possible indexes.

From MySQL 4.0.9 on, you can also use FORCE INDEX. This acts likes USE INDEX (*key_list*) but with the addition that a table scan is assumed to be *very* expensive. In other words, a table scan will only be used if there is no way to use one of the given indexes to find rows in the table.

USE KEY, IGNORE KEY, and FORCE KEY are synonyms for USE INDEX, IGNORE INDEX, and FORCE INDEX.

**Note:** USE INDEX, IGNORE INDEX, and FORCE INDEX only affect which indexes are used when MySQL decides how to find rows in the table and how to do the join. They do not affect whether an index will be used when resolving an ORDER BY or GROUP BY.

Some join 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;
mysql> SELECT * FROM table1 USE INDEX (key1,key2)
    ->          WHERE key1=1 AND key2=2 AND key3=3;
mysql> SELECT * FROM table1 IGNORE INDEX (key3)
    ->          WHERE key1=1 AND key2=2 AND key3=3;
```

### 6.1.7.2 UNION Syntax

```
SELECT ...
UNION [ALL | DISTINCT]
SELECT ...
  [UNION [ALL | DISTINCT]
   SELECT ...]
```

UNION is used to combine the result from many SELECT statements into one result set. UNION is available from MySQL 4.0.0 on.

Selected columns listed in corresponding positions of each SELECT statement should have the same type. (For example, the first column selected by the first statement should have the same type as the first column selected by the other statements.) The column names used in the first SELECT statement are used as the column names for the results returned.

The SELECT statements are normal select statements, but with the following restrictions:

- Only the last SELECT statement can have INTO OUTFILE.
- HIGH_PRIORITY cannot be used with SELECT statements that are part of a UNION. If you specify it for the first SELECT, it has no effect. If you specify it for any subsequent SELECT statements, a syntax error results.

If you don't use the keyword ALL for the UNION, all returned rows will be unique, as if you had done a DISTINCT for the total result set. If you specify ALL, you will get all matching rows from all the used SELECT statements.

The `DISTINCT` keyword is an optional word (introduced in MySQL 4.0.17). It does nothing, but is allowed in the syntax as required by the SQL standard.

**Note:** You cannot mix `UNION ALL` and `UNION DISTINCT` in the same query yet. If you use `ALL` for one `UNION` then it is used for all of them.

If you want to use an `ORDER BY` to sort the entire `UNION` result, you should use parentheses:

```
(SELECT a FROM tbl_name WHERE a=10 AND B=1 ORDER BY a LIMIT 10)
UNION
(SELECT a FROM tbl_name WHERE a=11 AND B=2 ORDER BY a LIMIT 10)
ORDER BY a;
```

The types and lengths of the columns in the result set of a `UNION` take into account the values retrieved by all the `SELECT` statements. Before MySQL 4.1.1, a limitation of `UNION` is that only the values from the first `SELECT` are used to determine result column types and lengths. This could result in value truncation if, for example, the first `SELECT` retrieves shorter values than the second `SELECT`:

```
mysql> SELECT REPEAT('a',1) UNION SELECT REPEAT('b',10);
+---------------+
| REPEAT('a',1) |
+---------------+
| a             |
| b             |
+---------------+
```

That limitation has been removed as of MySQL 4.1.1:

```
mysql> SELECT REPEAT('a',1) UNION SELECT REPEAT('b',10);
+---------------+
| REPEAT('a',1) |
+---------------+
| a             |
| bbbbbbbbbb    |
+---------------+
```

## 6.1.8 Subquery Syntax

A subquery is a `SELECT` statement inside another statement.

Starting with MySQL 4.1, all subquery forms and operations that the SQL standard requires are supported, as well as a few features that are MySQL-specific.

With earlier MySQL versions, it was necessary to work around or avoid the use of subqueries, but people starting to write code now will find that subqueries are a very useful part of the MySQL toolkit.

For MySQL versions prior to 4.1, most subqueries can be successfully rewritten using joins and other methods. See Section 6.1.8.11, "Rewriting Subqueries as Joins for Earlier MySQL Versions."

Here is an example of a subquery:

```
SELECT * FROM t1 WHERE column1 = (SELECT column1 FROM t2);
```

In this example, `SELECT * FROM t1 ...` is the *outer query* (or *outer statement*), and `(SELECT column1 FROM t2)` is the *subquery*. We say that the subquery is *nested* in the outer query, and in fact it's possible to nest subqueries within other subqueries, to a great depth. A subquery must always appear within parentheses.

The main advantages of subqueries are:

- They allow queries that are *structured* so that it's possible to isolate each part of a statement.
- They provide alternative ways to perform operations that would otherwise require complex joins and unions.
- They are, in many people's opinion, readable. Indeed, it was the innovation of subqueries that gave people the original idea of calling the early SQL "Structured Query Language."

Here is an example statement that shows the major points about subquery syntax as specified by the SQL standard and supported in MySQL:

```
DELETE FROM t1
WHERE s11 > ANY
 (SELECT COUNT(*) /* no hint */ FROM t2
 WHERE NOT EXISTS
  (SELECT * FROM t3
   WHERE ROW(5*t2.s1,77)=
    (SELECT 50,11*s1 FROM t4 UNION SELECT 50,77 FROM
     (SELECT * FROM t5) AS t5)));
```

### 6.1.8.1 The Subquery as Scalar Operand

In its simplest form (the *scalar* subquery as opposed to the *row* or *table* subqueries that are discussed later), a subquery is a simple operand. Thus, you can use it wherever a column value or literal is legal, and you can expect it to have those characteristics that all operands have: a data type, a length, an indication whether it can be `NULL`, and so on. For example:

```
CREATE TABLE t1 (s1 INT, s2 CHAR(5) NOT NULL);
SELECT (SELECT s2 FROM t1);
```

The subquery in this `SELECT` has a data type of `CHAR`, a length of 5, a character set and collation equal to the defaults in effect at `CREATE TABLE` time, and an indication that the value in the column can be `NULL`. In fact, almost all subqueries can be `NULL`, because if the table is empty, as in the example, the value of the subquery will be `NULL`. There are few restrictions.

- A subquery's outer statement can be any one of: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `SET`, or `DO`.
- A subquery can contain any of the keywords or clauses that an ordinary `SELECT` can contain: `DISTINCT`, `GROUP BY`, `ORDER BY`, `LIMIT`, joins, hints, `UNION` constructs, comments, functions, and so on.

So, when you see examples in the following sections that contain the rather spartan construct `(SELECT column1 FROM t1)`, imagine that your own code will contain much more diverse and complex constructions.

For example, suppose that we make two tables:

```
CREATE TABLE t1 (s1 INT);
INSERT INTO t1 VALUES (1);
CREATE TABLE t2 (s1 INT);
INSERT INTO t2 VALUES (2);
```

Then perform a `SELECT`:

```
SELECT (SELECT s1 FROM t2) FROM t1;
```

The result will be `2` because there is a row in `t2` containing a column `s1` that has a value of `2`.

The subquery can be part of an expression. If it is an operand for a function, don't forget the parentheses. For example:

```
SELECT UPPER((SELECT s1 FROM t1)) FROM t2;
```

### 6.1.8.2 Comparisons Using Subqueries

The most common use of a subquery is in the form:

```
non_subquery_operand comparison_operator (subquery)
```

Where *`comparison_operator`* is one of:

```
=  >  <  >=  <=  <>
```

For example:

```
... 'a' = (SELECT column1 FROM t1)
```

At one time the only legal place for a subquery was on the right side of a comparison, and you might still find some old DBMSs that insist on this.

Here is an example of a common-form subquery comparison that you cannot do with a join. It finds all the values in table `t1` that are equal to a maximum value in table `t2`:

```
SELECT column1 FROM t1
       WHERE column1 = (SELECT MAX(column2) FROM t2);
```

Here is another example, which again is impossible with a join because it involves aggregating for one of the tables. It finds all rows in table `t1` containing a value that occurs twice:

```
SELECT * FROM t1
       WHERE 2 = (SELECT COUNT(column1) FROM t1);
```

### 6.1.8.3 Subqueries with ANY, IN, and SOME

Syntax:

```
operand comparison_operator ANY (subquery)
operand IN (subquery)
operand comparison_operator SOME (subquery)
```

The `ANY` keyword, which must follow a comparison operator, means "return `TRUE` if the comparison is `TRUE` for `ANY` of the rows that the subquery returns." For example:

```
SELECT s1 FROM t1 WHERE s1 > ANY (SELECT s1 FROM t2);
```

Suppose that there is a row in table `t1` containing `(10)`. The expression is `TRUE` if table `t2` contains `(21,14,7)` because there is a value `7` in `t2` that is less than `10`. The expression is `FALSE` if table `t2` contains `(20,10)`, or if table `t2` is empty. The expression is `UNKNOWN` if table `t2` contains `(NULL,NULL,NULL)`.

The word `IN` is an alias for `= ANY`, so these two statements are the same:

```
SELECT s1 FROM t1 WHERE s1 = ANY (SELECT s1 FROM t2);
SELECT s1 FROM t1 WHERE s1 IN    (SELECT s1 FROM t2);
```

The word `SOME` is an alias for `ANY`, so these two statements are the same:

```
SELECT s1 FROM t1 WHERE s1 <> ANY  (SELECT s1 FROM t2);
SELECT s1 FROM t1 WHERE s1 <> SOME (SELECT s1 FROM t2);
```

Use of the word `SOME` is rare, but the preceding example shows why it might be useful. To most people's ears, the English phrase "a is not equal to any b" means "there is no b which is equal to a," but that isn't what is meant by the SQL syntax. Using `<> SOME` instead helps ensure that everyone understands the true meaning of the query.

### 6.1.8.4 Subqueries with ALL

Syntax:

```
operand comparison_operator ALL (subquery)
```

The word ALL, which must follow a comparison operator, means "return TRUE if the comparison is TRUE for ALL of the rows that the subquery returns." For example:

```
SELECT s1 FROM t1 WHERE s1 > ALL (SELECT s1 FROM t2);
```

Suppose that there is a row in table t1 containing (10). The expression is TRUE if table t2 contains (-5,0,+5) because 10 is greater than all three values in t2. The expression is FALSE if table t2 contains (12,6,NULL,-100) because there is a single value 12 in table t2 that is greater than 10. The expression is UNKNOWN if table t2 contains (0,NULL,1).

Finally, if table t2 is empty, the result is TRUE. You might think the result should be UNKNOWN, but sorry, it's TRUE. So, rather oddly, the following statement is TRUE when table t2 is empty:

```
SELECT * FROM t1 WHERE 1 > ALL (SELECT s1 FROM t2);
```

But this statement is UNKNOWN when table t2 is empty:

```
SELECT * FROM t1 WHERE 1 > (SELECT s1 FROM t2);
```

In addition, the following statement is UNKNOWN when table t2 is empty:

```
SELECT * FROM t1 WHERE 1 > ALL (SELECT MAX(s1) FROM t2);
```

In general, *tables with NULL values* and *empty tables* are *edge cases*. When writing subquery code, always consider whether you have taken those two possibilities into account.

### 6.1.8.5 Correlated Subqueries

A *correlated subquery* is a subquery that contains a reference to a column that also appears in the outer query. For example:

```
SELECT * FROM t1 WHERE column1 = ANY
       (SELECT column1 FROM t2 WHERE t2.column2 = t1.column2);
```

Notice that the subquery contains a reference to a column of t1, even though the subquery's FROM clause doesn't mention a table t1. So, MySQL looks outside the subquery, and finds t1 in the outer query.

Suppose that table t1 contains a row where column1 = 5 and column2 = 6; meanwhile, table t2 contains a row where column1 = 5 and column2 = 7. The simple expression ... WHERE column1 = ANY (SELECT column1 FROM t2) would be TRUE, but in this example, the WHERE clause within the subquery is FALSE (because 7 ≠ 5), so the subquery as a whole is FALSE.

**Scoping rule:** MySQL evaluates from inside to outside. For example:

```
SELECT column1 FROM t1 AS x
  WHERE x.column1 = (SELECT column1 FROM t2 AS x
    WHERE x.column1 = (SELECT column1 FROM t3
      WHERE x.column2 = t3.column1));
```

In this statement, `x.column2` must be a column in table `t2` because `SELECT column1 FROM t2 AS x ...` renames `t2`. It is not a column in table `t1` because `SELECT column1 FROM t1 ...` is an outer query that is *farther out*.

For subqueries in `HAVING` or `ORDER BY` clauses, MySQL also looks for column names in the outer select list.

For certain cases, a correlated subquery is optimized. For example:

```
val IN (SELECT key_val FROM tbl_name WHERE correlated_condition)
```

Otherwise, they are inefficient and likely to be slow. Rewriting the query as a join might improve performance.

### 6.1.8.6 EXISTS and NOT EXISTS

If a subquery returns any values at all, then `EXISTS subquery` is `TRUE`, and `NOT EXISTS subquery` is `FALSE`. For example:

```
SELECT column1 FROM t1 WHERE EXISTS (SELECT * FROM t2);
```

Traditionally, an `EXISTS` subquery starts with `SELECT *`, but it could begin with `SELECT 5` or `SELECT column1` or anything at all. MySQL ignores the `SELECT` list in such a subquery, so it doesn't matter.

For the preceding example, if `t2` contains any rows, even rows with nothing but `NULL` values, then the `EXISTS` condition is `TRUE`. This is actually an unlikely example, since almost always a `[NOT] EXISTS` subquery will contain correlations. Here are some more realistic examples:

- What kind of store is present in one or more cities?

```
SELECT DISTINCT store_type FROM Stores
  WHERE EXISTS (SELECT * FROM Cities_Stores
                WHERE Cities_Stores.store_type = Stores.store_type);
```

- What kind of store is present in no cities?

```
SELECT DISTINCT store_type FROM Stores
  WHERE NOT EXISTS (SELECT * FROM Cities_Stores
                    WHERE Cities_Stores.store_type = Stores.store_type);
```

- What kind of store is present in all cities?

```
SELECT DISTINCT store_type FROM Stores S1
  WHERE NOT EXISTS (
    SELECT * FROM Cities WHERE NOT EXISTS (
      SELECT * FROM Cities_Stores
       WHERE Cities_Stores.city = Cities.city
       AND Cities_Stores.store_type = Stores.store_type));
```

The last example is a double-nested `NOT EXISTS` query. That is, it has a `NOT EXISTS` clause within a `NOT EXISTS` clause. Formally, it answers the question "does a city exist with a store that is not in `Stores`?" But it's easier to say that a nested `NOT EXISTS` answers the question "is x TRUE for all y?"

## 6.1.8.7 Row Subqueries

The discussion to this point has been of *column (or scalar) subqueries*: subqueries that return a single column value. A *row subquery* is a subquery variant that returns a single row value and can thus return more than one column value. Here are two examples:

```
SELECT * FROM t1 WHERE (1,2) = (SELECT column1, column2 FROM t2);
SELECT * FROM t1 WHERE ROW(1,2) = (SELECT column1, column2 FROM t2);
```

The queries here are both `TRUE` if table `t2` has a row where `column1 = 1` and `column2 = 2`.

The expressions `(1,2)` and `ROW(1,2)` are sometimes called *row constructors*. The two are equivalent. They are legal in other contexts, too. For example, the following two statements are semantically equivalent (although currently only the second one can be optimized):

```
SELECT * FROM t1 WHERE (column1,column2) = (1,1);
SELECT * FROM t1 WHERE column1 = 1 AND column2 = 1;
```

The normal use of row constructors, though, is for comparisons with subqueries that return two or more columns. For example, the following query answers the request, "find all rows in table `t1` that also exist in table `t2`":

```
SELECT column1,column2,column3
       FROM t1
       WHERE (column1,column2,column3) IN
             (SELECT column1,column2,column3 FROM t2);
```

## 6.1.8.8 Subqueries in the `FROM` clause

Subqueries are legal in a `SELECT` statement's `FROM` clause. The syntax that you'll actually see is:

```
SELECT ... FROM (subquery) AS name ...
```

The `AS name` clause is mandatory, because every table in a `FROM` clause must have a name. Any columns in the *`subquery`* select list must have unique names.

For illustration, assume that you have this table:

```
CREATE TABLE t1 (s1 INT, s2 CHAR(5), s3 FLOAT);
```

Here's how to use a subquery in the `FROM` clause, using the example table:

```
INSERT INTO t1 VALUES (1,'1',1.0);
INSERT INTO t1 VALUES (2,'2',2.0);
SELECT sb1,sb2,sb3
       FROM (SELECT s1 AS sb1, s2 AS sb2, s3*2 AS sb3 FROM t1) AS sb
       WHERE sb1 > 1;
```

Result: `2, '2', 4.0`.

Here's another example: Suppose that you want to know the average of a set of sums for a grouped table. This won't work:

```
SELECT AVG(SUM(column1)) FROM t1 GROUP BY column1;
```

But this query will provide the desired information:

```
SELECT AVG(sum_column1)
       FROM (SELECT SUM(column1) AS sum_column1
             FROM t1 GROUP BY column1) AS t1;
```

Notice that the column name used within the subquery `(sum_column1)` is recognized in the outer query.

At the moment, subqueries in the `FROM` clause cannot be correlated subqueries.

Subquery in the `FROM` clause will be executed (that is, derived temporary tables will be built) even for the `EXPLAIN` statement, because upper level queries need information about all tables during optimization phase.

### 6.1.8.9 Subquery Errors

There are some new error returns that apply only to subqueries. This section groups them together because reviewing them will help remind you of some points.

- Unsupported subquery syntax:

  ```
  ERROR 1235 (ER_NOT_SUPPORTED_YET)
  SQLSTATE = 42000
  Message = "This version of MySQL doesn't yet support
  'LIMIT & IN/ALL/ANY/SOME subquery'"
  ```

  This means that statements of the following form will not work, although this happens only in some early versions, such as MySQL 4.1.1:

  ```
  SELECT * FROM t1 WHERE s1 IN (SELECT s2 FROM t2 ORDER BY s1 LIMIT 1)
  ```

- Incorrect number of columns from subquery:

  ```
  ERROR 1241 (ER_OPERAND_COLUMNF)
  SQLSTATE = 21000
  Message = "Operand should contain 1 column(s)"
  ```

  This error will occur in cases like this:

  ```
  SELECT (SELECT column1, column2 FROM t2) FROM t1;
  ```

  It's okay to use a subquery that returns multiple columns, if the purpose is comparison. See Section 6.1.8.7, "Row Subqueries." But in other contexts, the subquery must be a scalar operand.

- Incorrect number of rows from subquery:

  ```
  ERROR 1242 (ER_SUBSELECT_NO_1_ROW)
  SQLSTATE = 21000
  Message = "Subquery returns more than 1 row"
  ```

  This error will occur for statements such as the following one, but only when there is more than one row in `t2`:

  ```
  SELECT * FROM t1 WHERE column1 = (SELECT column1 FROM t2);
  ```

  That means this error might occur in code that had been working for years, because somebody happened to make a change that affected the number of rows that the subquery can return. Remember that if the object is to find any number of rows, not just one, then the correct statement would look like this:

  ```
  SELECT * FROM t1 WHERE column1 = ANY (SELECT column1 FROM t2);
  ```

- Incorrectly used table in subquery:

  ```
  Error 1093 (ER_UPDATE_TABLE_USED)
  SQLSTATE = HY000
  Message = "You can't specify target table 'x'
  for update in FROM clause"
  ```

  This error will occur in cases like this:

  ```
  UPDATE t1 SET column2 = (SELECT MAX(column1) FROM t1);
  ```

  It's okay to use a subquery for assignment within an `UPDATE` statement, since subqueries are legal in `UPDATE` and `DELETE` statements as well as in `SELECT` statements. However, you cannot use the same table, in this case table `t1`, for both the subquery's `FROM` clause and the update target.

Usually, failure of a subquery causes the entire statement to fail.

### 6.1.8.10 Optimizing Subqueries

Development is ongoing, so no optimization tip is reliable for the long term. Some interesting tricks that you might want to play with are:

- Use subquery clauses that affect the number or order of the rows in the subquery. For example:

```
SELECT * FROM t1 WHERE t1.column1 IN
  (SELECT column1 FROM t2 ORDER BY column1);
SELECT * FROM t1 WHERE t1.column1 IN
  (SELECT DISTINCT column1 FROM t2);
SELECT * FROM t1 WHERE EXISTS
  (SELECT * FROM t2 LIMIT 1);
```

- Replace a join with a subquery. For example, use this query:

```
SELECT DISTINCT column1 FROM t1 WHERE t1.column1 IN (
  SELECT column1 FROM t2);
```

  Instead of this query:

```
SELECT DISTINCT t1.column1 FROM t1, t2
  WHERE t1.column1 = t2.column1;
```

- Move clauses from outside to inside the subquery. For example, use this query:

```
SELECT * FROM t1
  WHERE s1 IN (SELECT s1 FROM t1 UNION ALL SELECT s1 FROM t2);
```

  Instead of this query:

```
SELECT * FROM t1
  WHERE s1 IN (SELECT s1 FROM t1) OR s1 IN (SELECT s1 FROM t2);
```

  For another example, use this query:

```
SELECT (SELECT column1 + 5 FROM t1) FROM t2;
```

  Instead of this query:

```
SELECT (SELECT column1 FROM t1) + 5 FROM t2;
```

- Use a row subquery instead of a correlated subquery. For example, use this query:

```
SELECT * FROM t1
  WHERE (column1,column2) IN (SELECT column1,column2 FROM t2);
```

  Instead of this query:

```
SELECT * FROM t1
  WHERE EXISTS (SELECT * FROM t2 WHERE t2.column1=t1.column1
  AND t2.column2=t1.column2);
```

- Use `NOT (a = ANY (...))` rather than `a <> ALL (...)`.
- Use `x = ANY (`*`table containing (1,2)`*`)` rather than `x=1 OR x=2`.
- Use `= ANY` rather than `EXISTS`.

These tricks might cause programs to go faster or slower. Using MySQL facilities like the `BENCHMARK()` function, you can get an idea about what helps in your own situation. Don't worry too much about transforming to joins except for compatibility with older versions of MySQL before 4.1 that do not support subqueries.

Some optimizations that MySQL itself makes are:

- MySQL executes non-correlated subqueries only once. Use `EXPLAIN` to make sure that a given subquery really is non-correlated.
- MySQL rewrites `IN`/`ALL`/`ANY`/`SOME` subqueries in an attempt to take advantage of the possibility that the select-list columns in the subquery are indexed.
- MySQL replaces subqueries of the following form with an index-lookup function, which `EXPLAIN` will describe as a special join type:

  ```
  ... IN (SELECT indexed_column FROM single_table ...)
  ```

- MySQL enhances expressions of the following form with an expression involving `MIN()` or `MAX()`, unless `NULL` values or empty sets are involved:

  ```
  value {ALL|ANY|SOME} {> | < | >= | <=} (non-correlated subquery)
  ```

  For example, this `WHERE` clause:

  ```
  WHERE 5 > ALL (SELECT x FROM t)
  ```

  might be treated by the optimizer like this:

  ```
  WHERE 5 > (SELECT MAX(x) FROM t)
  ```

There is a chapter titled "How MySQL Transforms Subqueries" in the MySQL Internals Manual. You can obtain this document by downloading the MySQL source package and looking for a file named `internals.texi` in the `Docs` directory.

## 6.1.8.11 Rewriting Subqueries as Joins for Earlier MySQL Versions

Before MySQL 4.1, only nested queries of the form `INSERT ... SELECT ...` and `REPLACE ... SELECT ...` are supported. The `IN()` construct can be used in other contexts to test membership in a set of values.

It is often possible to rewrite a query without a subquery:

```
SELECT * FROM t1 WHERE id IN (SELECT id FROM t2);
```

This can be rewritten as:

```
SELECT DISTINCT t1.* FROM t1,t2 WHERE t1.id=t2.id;
```

The queries:

```
SELECT * FROM t1 WHERE id NOT IN (SELECT id FROM t2);
SELECT * FROM t1 WHERE NOT EXISTS (SELECT id FROM t2 WHERE t1.id=t2.id);
```

Can be rewritten as:

```
SELECT table1.* FROM table1 LEFT JOIN table2 ON table1.id=table2.id
                                      WHERE table2.id IS NULL;
```

A `LEFT [OUTER] JOIN` can be faster than an equivalent subquery because the server might be able to optimize it better—a fact that is not specific to MySQL Server alone. Prior to SQL-92, outer joins did not exist, so subqueries were the only way to do certain things in those bygone days. Today, MySQL Server and many other modern database systems offer a whole range of outer join types.

For more complicated subqueries, you can often create temporary tables to hold the subquery. In some cases, however, this option will not work. The most frequently encountered of these cases arises with `DELETE` statements, for which standard SQL does not support joins (except in subqueries). For this situation, there are three options available:

- The first option is to upgrade to MySQL 4.1, which does support subqueries in `DELETE` statements.
- The second option is to use a procedural programming language (such as Perl or PHP) to submit a `SELECT` query to obtain the primary keys for the records to be deleted, and then use these values to construct the `DELETE` statement (`DELETE FROM ... WHERE` *`key_col`* `IN (`*`key1`*`,` *`key2`*`, ...)`).
- The third option is to use interactive SQL to construct a set of `DELETE` statements automatically, using the MySQL extension `CONCAT()` (in lieu of the standard `||` operator). For example:

  ```
  SELECT
    CONCAT('DELETE FROM table1 WHERE pkid = ', "'", table1.pkid, "'", ';')
    FROM table1, table2
   WHERE table1.column1 = table2.column2;
  ```

  You can place this query in a script file, use the file as input to one instance of the `mysql` program, and use the program output as input to a second instance of `mysql`:

  ```
  shell> mysql --skip-column-names mydb < myscript.sql | mysql mydb
  ```

MySQL Server 4.0 supports multiple-table `DELETE` statements that can be used to efficiently delete rows based on information from one table or even from many tables at the same time. Multiple-table `UPDATE` statements are also supported as of MySQL 4.0.

## 6.1.9 TRUNCATE Syntax

```
TRUNCATE TABLE tbl_name
```

`TRUNCATE TABLE` empties a table completely. Logically, this is equivalent to a `DELETE` statement that deletes all rows, but there are practical differences under some circumstances.

For `InnoDB`, `TRUNCATE TABLE` is mapped to `DELETE`, so there is no difference. For other storage engines, `TRUNCATE TABLE` differs from `DELETE FROM ...` in the following ways from MySQL 4.0 and up:

- Truncate operations drop and re-create the table, which is much faster than deleting rows one by one.
- Truncate operations are not transaction-safe; you will get an error if you have an active transaction or an active table lock.
- The number of deleted rows is not returned.
- As long as the table definition file *`tbl_name`*`.frm` is valid, the table can be re-created as an empty table with `TRUNCATE TABLE`, even if the data or index files have become corrupted.
- The table handler does not remember the last used `AUTO_INCREMENT` value, but starts counting from the beginning. This is true even for `MyISAM`, which normally does not reuse sequence values.

In MySQL 3.23, `TRUNCATE TABLE` is mapped to `COMMIT; DELETE FROM` *`tbl_name`*, so it behaves like `DELETE`. See Section 6.1.1, "`DELETE` Syntax."

`TRUNCATE TABLE` is an Oracle SQL extension. This statement was added in MySQL 3.23.28, although from 3.23.28 to 3.23.32, the keyword `TABLE` must be omitted.

## 6.1.10 UPDATE Syntax

Single-table syntax:

```
UPDATE [LOW_PRIORITY] [IGNORE] tbl_name
    SET col_name1=expr1 [, col_name2=expr2 ...]
    [WHERE where_definition]
    [ORDER BY ...]
    [LIMIT row_count]
```

Multiple-table syntax:

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

The UPDATE statement 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 the ORDER BY clause is specified, the rows will be updated in the order that is specified. The LIMIT clause places a limit on the number of rows that can be updated.

The UPDATE statement supports the following modifiers:

- If you specify the LOW_PRIORITY keyword, execution of the UPDATE is delayed until no other clients are reading from the table.
- If you specify the IGNORE keyword, the update statement will not abort even if duplicate-key errors occur during the update. Rows for which conflicts occur are not updated.

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.

If you update a column that has been declared NOT NULL by setting to NULL, the column is set to the default value appropriate for the column type and the warning count is incremented. The default value is 0 for numeric types, the empty string ('') for string types, and the "zero" value for date and time types.

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

Starting from MySQL 3.23, you can use LIMIT *row_count* to restrict the scope of the UPDATE. A LIMIT clause works as follows:

- Before MySQL 4.0.13, LIMIT is a rows-affected restriction. The statement stops as soon as it has changed *row_count* rows that satisfy the WHERE clause.
- From 4.0.13 on, LIMIT is a rows-matched restriction. The statement stops as soon as it has found *row_count* rows that satisfy the WHERE clause, whether or not they actually were changed.

If an UPDATE statement includes an ORDER BY clause, the rows are updated in the order specified by the clause. ORDER BY can be used from MySQL 4.0.0.

Starting with MySQL 4.0.4, you can also perform UPDATE operations that cover multiple tables:

```
UPDATE items,month SET items.price=month.price
WHERE items.id=month.id;
```

The example shows an inner join using the comma operator, but multiple-table UPDATE statements can use any type of join allowed in SELECT statements, such as LEFT JOIN.

**Note:** You cannot use ORDER BY or LIMIT with multiple-table UPDATE.

Before MySQL 4.0.18, you need the UPDATE privilege for all tables used in a multiple-table UPDATE, even if they were not updated. As of MySQL 4.0.18, you need only the SELECT privilege for any columns that are read but not modified.

If you use a multiple-table UPDATE statement involving InnoDB tables for which there are foreign key constraints, the MySQL optimizer might process tables in an order that differs from that of their parent/child relationship. In this case, the statement will fail and roll back. Instead, update a single table and rely on the ON UPDATE capabilities that InnoDB provides to cause the other tables to be modified accordingly.

## 6.2 Data Definition Statements

### 6.2.1 ALTER DATABASE Syntax

```
ALTER DATABASE db_name
    alter_specification [, alter_specification] ...

alter_specification:
    [DEFAULT] CHARACTER SET charset_name
  | [DEFAULT] COLLATE collation_name
```

ALTER DATABASE allows you to change the overall characteristics of a database. These characteristics are stored in the db.opt file in the database directory. To use ALTER DATABASE, you need the ALTER privilege on the database.

The CHARACTER SET clause changes the default database character set. The COLLATE clause changes the default database collation. Character set and collation names are discussed in Chapter 3, "Character Set Support."

ALTER DATABASE was added in MySQL 4.1.1.

### 6.2.2 ALTER TABLE Syntax

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

alter_specification:
    ADD [COLUMN] column_definition [FIRST | AFTER col_name ]
  | ADD [COLUMN] (column_definition,...)
  | ADD INDEX [index_name] [index_type] (index_col_name,...)
  | ADD [CONSTRAINT [symbol]]
        PRIMARY KEY [index_type] (index_col_name,...)
  | ADD [CONSTRAINT [symbol]]
        UNIQUE [index_name] [index_type] (index_col_name,...)
  | ADD [FULLTEXT|SPATIAL] [index_name] (index_col_name,...)
  | ADD [CONSTRAINT [symbol]]
        FOREIGN KEY [index_name] (index_col_name,...)
        [reference_definition]
  | ALTER [COLUMN] col_name {SET DEFAULT literal | DROP DEFAULT}
  | CHANGE [COLUMN] old_col_name column_definition
        [FIRST|AFTER col_name]
  | MODIFY [COLUMN] column_definition [FIRST | AFTER col_name]
  | DROP [COLUMN] col_name
  | DROP PRIMARY KEY
  | DROP INDEX index_name
  | DROP FOREIGN KEY fk_symbol
  | DISABLE KEYS
  | ENABLE KEYS
  | RENAME [TO] new_tbl_name
  | ORDER BY col_name
  | CONVERT TO CHARACTER SET charset_name [COLLATE collation_name]
  | [DEFAULT] CHARACTER SET charset_name [COLLATE collation_name]
  | DISCARD TABLESPACE
  | IMPORT TABLESPACE
  | table_options
```

`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.

The syntax for many of the allowable alterations is similar to clauses of the `CREATE TABLE` statement. See Section 6.2.5, "`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 6.2.5.2, "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. 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, then are automatically redirected to the new table without any failed updates.

Note that if you use any other option to ALTER TABLE than RENAME, MySQL always creates a temporary table, even if the data wouldn't strictly need to be copied (such as when you change the name of a column). We plan to fix this in the future, but because ALTER TABLE is not a statement that is normally used frequently, this isn't high on our TODO list. For MyISAM tables, you can speed up the index re-creation operation (which is the slowest part of the alteration process) by setting the myisam_sort_buffer_size system variable to a high value.

- To use ALTER TABLE, you need ALTER, INSERT, and CREATE privileges for the table.
- IGNORE is a MySQL extension to standard SQL. 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 duplicate-key errors occur. 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 standard SQL, which allows only one of each clause per ALTER TABLE statement.
- CHANGE *col_name*, DROP *col_name*, and DROP INDEX are MySQL extensions to standard SQL.
- MODIFY is an Oracle extension to ALTER TABLE.
- The word COLUMN is purely optional and can be omitted.
- If you use ALTER TABLE *tbl_name* RENAME TO *new_tbl_name* without any other options, MySQL simply renames any files that correspond to the table *tbl_name*. There is no need to create a temporary table. (You can also use the RENAME TABLE statement to rename tables. See Section 6.2.9, "RENAME TABLE Syntax.")
- *column_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 6.2.5, "CREATE TABLE Syntax."
- You can rename a column using a CHANGE *old_col_name column_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 an old and a new column name, 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 example, 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 existing column values to the new type as well as possible.
- In MySQL 3.22 or later, you can use FIRST or AFTER *col_name* to add a column at a specific position within a table row. The default is to add the column last. From MySQL 4.0.1 on, you can also use FIRST and AFTER in CHANGE or MODIFY operations.
- 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, as described in Section 6.2.5, "CREATE TABLE Syntax."
- DROP INDEX removes an index. This is a MySQL extension to standard SQL. See Section 6.2.7, "DROP INDEX Syntax."
- 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.
- If a table contains only one column, the column cannot be dropped. If what you intend is to remove the table, use DROP TABLE instead.
- DROP PRIMARY KEY drops the primary index. (Prior to MySQL 4.1.2, if no primary index exists, DROP PRIMARY KEY 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.)

  If you add a UNIQUE INDEX or PRIMARY KEY to a table, it is stored before any non-unique index so that MySQL can detect duplicate keys as early as possible.
- ORDER BY allows you to create the new table with the rows in a specific order. Note that the table will not remain in this order after inserts and deletes. This option is mainly useful when you know that you are mostly going to query the rows in a certain order; by using this option after big changes to the table, you might be able to get higher performance. In some cases, it might make sorting easier for MySQL if the table is in order by the column that you want to order it by later.
- If you use ALTER TABLE on a MyISAM table, all non-unique indexes are created in a separate batch (as for REPAIR TABLE). This should make ALTER TABLE much faster when you have many indexes.

As of MySQL 4.0, this feature can be activated explicitly. `ALTER TABLE ... DISABLE KEYS` tells MySQL to stop updating non-unique indexes for a `MyISAM` table. `ALTER TABLE ... ENABLE KEYS` then should be used to re-create missing indexes. MySQL does this with a special algorithm that is much faster than inserting keys one by one, so disabling keys before performing bulk insert operations should give a considerable speedup.

- The `FOREIGN KEY` and `REFERENCES` clauses are supported by the `InnoDB` storage engine, which implements `ADD [CONSTRAINT [symbol]] FOREIGN KEY (...) REFERENCES ... (...)`. For other storage engines, the clauses are parsed but ignored. The `CHECK` clause is parsed but ignored by all storage engines. See Section 6.2.5, "`CREATE TABLE` Syntax." The reason for accepting but ignoring syntax clauses is for compatibility, to make it easier to port code from other SQL servers, and to run applications that create tables with references. See Section 1.8.5, "MySQL Differences from Standard SQL."
- Starting from MySQL 4.0.13, `InnoDB` supports the use of `ALTER TABLE` to drop foreign keys:

```
ALTER TABLE yourtablename DROP FOREIGN KEY fk_symbol:
```

- `ALTER TABLE` ignores the `DATA DIRECTORY` and `INDEX DIRECTORY` table options.
- From MySQL 4.1.2 on, if you want to change all character columns (`CHAR`, `VARCHAR`, `TEXT`) to a new character set, use a statement like this:

```
ALTER TABLE tbl_name CONVERT TO CHARACTER SET charset_name;
```

This is useful, for example, after upgrading from MySQL 4.0.x to 4.1.x. See Section 3.10, "Upgrading Character Sets from MySQL 4.0."

**Warning:** The preceding operation will convert column values between the character sets. This is *not* what you want if you have a column in one character set (like `latin1`) but the stored values actually use some other, incompatible character set (like `utf8`). In this case, you have to do the following for each such column:

```
ALTER TABLE t1 CHANGE c1 c1 BLOB;
ALTER TABLE t1 CHANGE c1 c1 TEXT CHARACTER SET utf8;
```

The reason this works is that there is no conversion when you convert to or from `BLOB` columns.

To change only the *default* character set for a table, use this statement:

```
ALTER TABLE tbl_name DEFAULT CHARACTER SET charset_name;
```

The word `DEFAULT` is optional. The default character set is the character set that is used if you don't specify the character set for a new column you add to a table (for example, with `ALTER TABLE ... ADD column`).

**Warning:** From MySQL 4.1.2 and up, `ALTER TABLE ... DEFAULT CHARACTER SET` and `ALTER TABLE ... CHARACTER SET` are equivalent and change only the default table character set. In MySQL 4.1 releases before 4.1.2, `ALTER TABLE ... DEFAULT CHARACTER SET` changes the default character set, but `ALTER TABLE ... CHARACTER SET` (without `DEFAULT`) changes the default character set *and* also converts all columns to the new character set.

- For an `InnoDB` table that is created with its own tablespace in an `.ibd` file, that file can be discarded and imported. To discard the `.ibd` file, use this statement:

  ```
  ALTER TABLE tbl_name DISCARD TABLESPACE;
  ```

  This deletes the current `.ibd` file, so be sure that you have a backup first. Attempting to access the table while the tablespace file is discarded results in an error.

  To import the backup `.ibd` file back into the table, copy it into the database directory, then issue this statement:

  ```
  ALTER TABLE tbl_name IMPORT TABLESPACE;
  ```

- With the `mysql_info()` C API function, 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.

Here are some examples that show uses of `ALTER TABLE`. Begin with a table `t1` that is created as shown here:

```
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 indexes on column `d` and on column `a`:

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

Note that we indexed `c` (as a `PRIMARY KEY`), because `AUTO_INCREMENT` columns must be indexed, and also that we declare `c` as `NOT NULL`, because primary key columns cannot be `NULL`.

When you add an `AUTO_INCREMENT` column, column values are filled in with sequence numbers for you automatically. For `MyISAM` tables, you can set the first sequence number by executing `SET INSERT_ID=`*`value`* before `ALTER TABLE` or by using the `AUTO_INCREMENT=`*`value`* table option. See Section 6.5.3.1, "`SET` Syntax."

With `MyISAM` tables, if you don't change the `AUTO_INCREMENT` column, the sequence number will not be affected. If you drop an `AUTO_INCREMENT` column and then add another `AUTO_INCREMENT` column, the numbers are resequenced beginning with 1.

Starting from MySQL 3.23.50, `InnoDB` allows you to add a new foreign key constraint to a table by using `ALTER TABLE`:

```
ALTER TABLE yourtablename
    ADD [CONSTRAINT symbol] FOREIGN KEY [id] (index_col_name, ...)
    REFERENCES tbl_name (index_col_name, ...)
    [ON DELETE {CASCADE | SET NULL | NO ACTION | RESTRICT}]
    [ON UPDATE {CASCADE | SET NULL | NO ACTION | RESTRICT}]
```

**Remember to create the required indexes first**. You can also add a self-referential foreign key constraint to a table using `ALTER TABLE`.

Starting from MySQL 4.0.13, `InnoDB` supports the use of `ALTER TABLE` to drop foreign keys:

```
ALTER TABLE yourtablename DROP FOREIGN KEY fk_symbol:
```

If the `FOREIGN KEY` clause included a `CONSTRAINT` name when you created the foreign key, you can refer to that name to drop the foreign key. (A constraint name can be given as of MySQL 4.0.18.) Otherwise, the *`fk_symbol`* value is internally generated by `InnoDB` when the foreign key is created. To find out the symbol when you want to drop a foreign key, use the `SHOW CREATE TABLE` statement. An example:

```
mysql> SHOW CREATE TABLE ibtest11c\G
*************************** 1. row ***************************
       Table: ibtest11c
Create Table: CREATE TABLE `ibtest11c` (
  `A` int(11) NOT NULL auto_increment,
  `D` int(11) NOT NULL default '0',
  `B` varchar(200) NOT NULL default '',
  `C` varchar(175) default NULL,
  PRIMARY KEY  (`A`,`D`,`B`),
  KEY `B` (`B`,`C`),
  KEY `C` (`C`),
  CONSTRAINT `0_38775` FOREIGN KEY (`A`, `D`)
REFERENCES `ibtest11a` (`A`, `D`)
ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `0_38776` FOREIGN KEY (`B`, `C`)
REFERENCES `ibtest11a` (`B`, `C`)
ON DELETE CASCADE ON UPDATE CASCADE
) TYPE=InnoDB CHARSET=latin1
1 row in set (0.01 sec)

mysql> ALTER TABLE ibtest11c DROP FOREIGN KEY 0_38775;
```

Before MySQL 3.23.50, `ALTER TABLE` or `CREATE INDEX` should not be used in connection with tables that have foreign key constraints or that are referenced in foreign key constraints: Any `ALTER TABLE` removes all foreign key constraints defined for the table. You should not use `ALTER TABLE` with the referenced table, either. Instead, use `DROP TABLE` and `CREATE TABLE` to modify the schema. When MySQL does an `ALTER TABLE` it may internally use `RENAME TABLE`, and that will confuse the foreign key constraints that refer to the table. In MySQL, a `CREATE INDEX` statement is processed as an `ALTER TABLE`, so the same considerations apply.

See Section A.3.1, "Problems with `ALTER TABLE`."

## 6.2.3 CREATE DATABASE Syntax

```
CREATE DATABASE [IF NOT EXISTS] db_name
    [create_specification [, create_specification] ...]

create_specification:
    [DEFAULT] CHARACTER SET charset_name
  | [DEFAULT] COLLATE collation_name
```

`CREATE DATABASE` creates a database with the given name. To use `CREATE DATABASE`, you need the `CREATE` privilege on the database.

Rules for allowable database names are given in Section 2.2, "Database, Table, Index, Column, and Alias Names." An error occurs if the database already exists and you didn't specify `IF NOT EXISTS`.

As of MySQL 4.1.1, *`create_specification`* options can be given to specify database characteristics. Database characteristics are stored in the `db.opt` file in the database directory. The `CHARACTER SET` clause specifies the default database character set. The `COLLATE` clause specifies the default database collation. Character set and collation names are discussed in Chapter 3, "Character Set Support."

Databases in MySQL are implemented as directories containing files that correspond to tables in the database. Because 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 (and the `db.opt` file, for MySQL 4.1.1 and up).

You can also use the `mysqladmin` program to create databases.

## 6.2.4 `CREATE INDEX` Syntax

```
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name [index_type]
    ON tbl_name (index_col_name,...)

index_col_name:
    col_name [(length)] [ASC | DESC]
```

In MySQL 3.22 or later, `CREATE INDEX` is mapped to an `ALTER TABLE` statement to create indexes. See Section 6.2.2, "`ALTER TABLE` Syntax." The `CREATE INDEX` statement doesn't do anything prior to MySQL 3.22.

Normally, you create all indexes on a table at the time the table itself is created with `CREATE TABLE`. See Section 6.2.5, "`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 to index a prefix consisting of the first *length* characters of each column value. `BLOB` and `TEXT` columns also can be indexed, but a prefix length *must* be given.

The statement shown here creates an index using the first 10 characters of the `name` column:

```
CREATE INDEX part_of_name ON customer (name(10));
```

Because 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!

Prefixes can be up to 255 bytes long (or 1000 bytes for `MyISAM` and `InnoDB` tables as of MySQL 4.1.2). Note that prefix limits are measured in bytes, whereas the prefix length in `CREATE INDEX` statements is interpreted as number of characters. Take this into account when specifying a prefix length for a column that uses a multi-byte character set.

Note that you can add an index on a column that can have `NULL` values only if you are using MySQL 3.23.2 or newer and are using the `MyISAM`, `InnoDB`, or `BDB` table type. You can only add an index on a `BLOB` or `TEXT` column if you are using MySQL 3.23.2 or newer and are using the `MyISAM` or `BDB` table type, or MySQL 4.0.14 or newer and the `InnoDB` table type.

An *index_col_name* specification can end with `ASC` or `DESC`. These keywords are allowed for future extensions for specifying ascending or descending index value storage. Currently, they are parsed but ignored; index values are always stored in ascending order.

`FULLTEXT` indexes can index only `CHAR`, `VARCHAR`, and `TEXT` columns, and only in `MyISAM` tables. `FULLTEXT` indexes are available in MySQL 3.23.23 and later. Section 5.6, "Full-Text Search Functions."

SPATIAL indexes can index only spatial columns, and only in MyISAM tables. SPATIAL indexes are available in MySQL 4.1 or later. Spatial column types are described in Chapter 7, "Spatial Extensions in MySQL."

### 6.2.5 CREATE TABLE Syntax

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

Or:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [(] LIKE old_tbl_name [)];

create_definition:
    column_definition
  | [CONSTRAINT [symbol]] PRIMARY KEY [index_type] (index_col_name,...)
  | KEY [index_name] [index_type] (index_col_name,...)
  | INDEX [index_name] [index_type] (index_col_name,...)
  | [CONSTRAINT [symbol]] UNIQUE [INDEX]
        [index_name] [index_type] (index_col_name,...)
  | [FULLTEXT|SPATIAL] [INDEX] [index_name] (index_col_name,...)
  | [CONSTRAINT [symbol]] FOREIGN KEY
        [index_name] (index_col_name,...) [reference_definition]
  | CHECK (expr)

column_definition:
    col_name type [NOT NULL | NULL] [DEFAULT default_value]
        [AUTO_INCREMENT] [[PRIMARY] KEY] [COMMENT 'string']
        [reference_definition]

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

index_col_name:
    col_name [(length)] [ASC | DESC]

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: table_option [table_option] ...

table_option:
    {ENGINE|TYPE} = {BDB|HEAP|ISAM|InnoDB|MERGE|MRG_MYISAM|MYISAM}
  | AUTO_INCREMENT = value
  | AVG_ROW_LENGTH = value
  | CHECKSUM = {0 | 1}
  | COMMENT = 'string'
  | MAX_ROWS = value
  | MIN_ROWS = value
  | PACK_KEYS = {0 | 1 | DEFAULT}
  | PASSWORD = 'string'
  | DELAY_KEY_WRITE = {0 | 1}
  | ROW_FORMAT = { DEFAULT | DYNAMIC | FIXED | COMPRESSED }
  | RAID_TYPE = { 1 | STRIPED | RAID0 }
        RAID_CHUNKS = value
        RAID_CHUNKSIZE = value
  | UNION = (tbl_name[,tbl_name]...)
  | INSERT_METHOD = { NO | FIRST | LAST }
  | DATA DIRECTORY = 'absolute path to directory'
  | INDEX DIRECTORY = 'absolute path to directory'
  | [DEFAULT] CHARACTER SET charset_name [COLLATE collation_name]

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

`CREATE TABLE` creates a table with the given name. You must have the `CREATE` privilege for the table.

Rules for allowable table names are given in Section 2.2, "Database, Table, Index, Column, and Alias Names." By default, the table is created in the current database. An error occurs if the table already exists, if there is no current database, or if the database does not exist.

In MySQL 3.22 or later, the table name can be specified as *`db_name.tbl_name`* to create the table in a specific database. This works whether or not there is a current database. If you use quoted identifiers, quote the database and table names separately. For example, `` `mydb`.`mytbl` `` is legal, but `` `mydb.mytbl` `` is not.

From MySQL 3.23 on, you can use the `TEMPORARY` keyword when creating a table. A `TEMPORARY` table is visible only to the current connection, and is dropped automatically when the connection is closed. This means that two different connections can use the same temporary table name without conflicting with each other or with an existing non-`TEMPORARY` table of the same name. (The existing table is hidden until the temporary table is dropped.) From MySQL 4.0.2 on, you must have the `CREATE TEMPORARY TABLES` privilege to be able to create temporary tables.

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 existing table has a structure identical to that indicated by the `CREATE TABLE` statement.

MySQL represents each table by an `.frm` table format (definition) file in the database directory. The storage engine for the table might create other files as well. In the case of `MyISAM` tables, the storage engine creates three files for a table named *`tbl_name`*:

| **File** | **Purpose** |
|---|---|
| *tbl_name*.frm | Table format (definition) file |
| *tbl_name*.MYD | Data file |
| *tbl_name*.MYI | Index file |

The files created by each storage engine to represent tables are described in the *MySQL Administrator's Guide*.

For general information on the properties of the various column types, see Chapter 4, "Column Types." For information about spatial column types, see Chapter 7, "Spatial Extensions in MySQL."

- If neither `NULL` nor `NOT NULL` is specified, the column is treated as though `NULL` had been specified.
- An integer column can have the additional attribute `AUTO_INCREMENT`. When you insert a value of `NULL` (recommended) or `0` into an indexed `AUTO_INCREMENT` column, the column is set to the next sequence value. Typically, this is *`value`*`+1`, where *`value`* is the largest value for the column currently in the table. `AUTO_INCREMENT` sequences begin with `1`.

  As of MySQL 4.1.1, specifying the `NO_AUTO_VALUE_ON_ZERO` flag for the `--sql-mode` server option or the `sql_mode` system variable allows you to store `0` in `AUTO_INCREMENT` columns as `0` without generating a new sequence value.

  **Note:** There can be only one `AUTO_INCREMENT` column per table, it must be indexed, and it cannot have a `DEFAULT` value. As of MySQL 3.23, an `AUTO_INCREMENT` column will work properly only if it contains only positive values. Inserting a negative number is regarded as inserting a very large positive number. This is done to avoid precision problems when numbers "wrap" over from positive to negative and also to ensure that you don't accidentally get an `AUTO_INCREMENT` column that contains `0`.

  For `MyISAM` and `BDB` tables, you can specify an `AUTO_INCREMENT` secondary column in a multiple-column key.

  To make MySQL compatible with some ODBC applications, you can find the `AUTO_INCREMENT` value for the last inserted row with the following query:

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

- As of MySQL 4.1, character column definitions can include a `CHARACTER SET` attribute to specify the character set and, optionally, a collation for the column. For details, see Chapter 3, "Character Set Support."

  ```
  CREATE TABLE t (c CHAR(20) CHARACTER SET utf8 COLLATE utf8_bin);
  ```

  Also as of 4.1, MySQL interprets length specifications in character column definitions in characters. (Earlier versions interpret them in bytes.)

- `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 can 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.

- A `DEFAULT` value must be a constant; it cannot be a function or an expression. This means, for example, that you cannot set the default for a date column to be the value of a function such as `NOW()` or `CURRENT_DATE`.

  If no `DEFAULT` value is specified for a column, MySQL automatically assigns one, as follows.

  If the column can 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 4.3, "Date and Time Types."
  - For string types other than `ENUM`, the default value is the empty string. For `ENUM`, the default is the first enumeration value.

  `BLOB` and `TEXT` columns cannot be assigned a default value.

- A comment for a column can be specified with the `COMMENT` option. The comment is displayed by the `SHOW CREATE TABLE` and `SHOW FULL COLUMNS` statements. This option is operational as of MySQL 4.1. (It is allowed but ignored in earlier versions.)
- From MySQL 4.1.0 on, the attribute `SERIAL` can be used as an alias for `BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE`. This is a compatibility feature.
- `KEY` is normally a synonym for `INDEX`. From MySQL 4.1, the key attribute `PRIMARY KEY` can also be specified as just `KEY` when given in a column definition. This was implemented for compatibility with other database systems.
- In MySQL, a `UNIQUE` index is one in which all values in the index must be distinct. An error occurs if you try to add a new row with a key that matches an existing row. The exception to this is that if a column in the index is allowed to contain `NULL` values, it can contain multiple `NULL` values. This exception does not apply to `BDB` tables, for which indexed columns allow only a single `NULL`.
- A `PRIMARY KEY` is a unique `KEY` where all key columns must be defined as `NOT NULL`. If they are not explicitly declared as `NOT NULL`, MySQL will declare them so implicitly (and silently). A table can have only one `PRIMARY KEY`. If you don't have a `PRIMARY KEY` and an application asks for the `PRIMARY KEY` in your tables, MySQL returns the first `UNIQUE` index that has no `NULL` columns as the `PRIMARY KEY`.
- In the created table, a `PRIMARY KEY` is placed first, followed by all `UNIQUE` indexes, and then the non-unique indexes. This helps the MySQL optimizer to prioritize which index to use and also more quickly to detect duplicated `UNIQUE` keys.

- 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 a separate `PRIMARY KEY(`*`index_col_name`*`, ...)` clause.
- If a `PRIMARY KEY` or `UNIQUE` index consists of only one column that has an integer type, you can also refer to the column as `_rowid` in `SELECT` statements (new in MySQL 3.23.11).
- In MySQL, the name of a `PRIMARY KEY` is `PRIMARY`. For other indexes, if you don't assign a name, the index is assigned the same name as the first indexed column, 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 6.5.3.7, "`SHOW DATABASES` Syntax."
- From MySQL 4.1.0 on, some storage engines allow you to specify an index type when creating an index. The syntax for the *`index_type`* specifier is `USING` *`type_name`*. The allowable *`type_name`* values supported by different storage engines are shown in the following table. Where multiple index types are listed, the first one is the default when no *`index_type`* specifier is given.

  | Storage Engine | Allowable Index Types |
  |---|---|
  | MyISAM | BTREE |
  | InnoDB | BTREE |
  | MEMORY/HEAP | HASH, BTREE |

  Example:

  ```
  CREATE TABLE lookup
      (id INT, INDEX USING BTREE (id))
      ENGINE = MEMORY;
  ```

  `TYPE` *`type_name`* can be used as a synonym for `USING` *`type_name`* to specify an index type. However, `USING` is the preferred form. Also, the index name that precedes the index type in the index specification syntax is not optional with `TYPE`. This is because, unlike `USING`, `TYPE` is not a reserved word and thus is interpreted as an index name.

  If you specify an index type that is not legal for a storage engine, but there is another index type available that the engine can use without affecting query results, the engine will use the available type.
- Only the `MyISAM`, `InnoDB`, `BDB`, and (as of MySQL 4.0.2) `MEMORY` storage engines support indexes on columns that can have `NULL` values. In other cases, you must declare indexed columns as `NOT NULL` or an error results.
- With *`col_name`*`(`*`length`*`)` syntax in an index specification, you can create an index that uses only the first *`length`* bytes of a `CHAR` or `VARCHAR` column. Indexing only a prefix of column values like this can make the index file much smaller.

The MyISAM and (as of MySQL 4.0.14) InnoDB storage engines also support indexing on BLOB and TEXT columns. When indexing a BLOB or TEXT column, you *must* specify a prefix length for the index. For example:

```
CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));
```

Prefixes can be up to 255 bytes long (or 1000 bytes for MyISAM and InnoDB tables as of MySQL 4.1.2). Note that prefix limits are measured in bytes, whereas the prefix length in CREATE TABLE statements is interpreted as number of characters. Take this into account when specifying a prefix length for a column that uses a multi-byte character set.

- An *index_col_name* specification can end with ASC or DESC. These keywords are allowed for future extensions for specifying ascending or descending index value storage. Currently, they are parsed but ignored; index values are always stored in ascending order.
- When you use ORDER BY or GROUP BY with a TEXT or BLOB column, the server sorts values using only the initial number of bytes indicated by the max_sort_length system variable. See Section 4.4.2, "The BLOB and TEXT Types."
- In MySQL 3.23.23 or later, you can create special FULLTEXT indexes. They are used for full-text search. Only the MyISAM table type supports FULLTEXT indexes. They can be created only from CHAR, VARCHAR, and TEXT columns. Indexing always happens over the entire column; partial indexing is not supported and any prefix length is ignored if specified. See Section 5.6, "Full-Text Search Functions," for details of operation.
- In MySQL 4.1 or later, you can create special SPATIAL indexes on spatial column types. Spatial types are supported only for MyISAM tables and indexed columns must be declared as NOT NULL. See Chapter 7, "Spatial Extensions in MySQL."
- In MySQL 3.23.44 or later, InnoDB tables support checking of foreign key constraints. Note that the FOREIGN KEY syntax in InnoDB is more restrictive than the syntax presented for the CREATE TABLE statement at the beginning of this section: The columns of the referenced table must always be explicitly named. InnoDB supports both ON DELETE and ON UPDATE actions on foreign keys as of MySQL 3.23.50 and 4.0.8, respectively. For the precise syntax, see Section 6.2.5.1, "Creating Foreign Keys."

  For other storage engines, MySQL Server parses the FOREIGN KEY and REFERENCES syntax in CREATE TABLE statements, but without further action being taken. The CHECK clause is parsed but ignored by all storage engines.
- For MyISAM and ISAM tables, 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 + delete_flag + 7)/8
             + (number of variable-length columns)
```

*delete_flag* is 1 for tables with static record format. Static tables use a bit in the row record for a flag that indicates whether the row has been deleted. *delete_flag* is 0 for dynamic tables because the flag is stored in the dynamic row header.

These calculations do not apply for InnoDB tables, for which storage size is no different for NULL columns than for NOT NULL columns.

The *table_options* part of the CREATE TABLE syntax can be used in MySQL 3.23 and above.

The ENGINE and TYPE options specify the storage engine for the table. ENGINE was added in MySQL 4.0.18 (for 4.0) and 4.1.2 (for 4.1). It is the preferred option name as of those versions, and TYPE has become deprecated. TYPE will be supported throughout the 4.x series, but likely will be removed in MySQL 5.1.

The ENGINE and TYPE options take the following values:

| Storage Engine | Description |
|---|---|
| BDB | Transaction-safe tables with page locking. |
| BerkeleyDB | An alias for BDB. |
| HEAP | The data for this table is stored only in memory. |
| ISAM | The original MySQL storage engine. |
| InnoDB | Transaction-safe tables with row locking and foreign keys. |
| MEMORY | An alias for HEAP. (Actually, as of MySQL 4.1, MEMORY is the preferred term.) |
| MERGE | A collection of MyISAM tables used as one table. |
| MRG_MyISAM | An alias for MERGE. |
| MyISAM | The binary portable storage engine that is the improved replacement for ISAM. |

If a storage engine is specified that is not available, MySQL uses MyISAM instead. For example, if a table definition includes the ENGINE=BDB option but the MySQL server does not support BDB tables, the table is created as a MyISAM table. This makes it possible to have a replication setup where you have transactional tables on the master but tables created on the slave are non-transactional (to get more speed). In MySQL 4.1.1, a warning occurs if the storage engine specification is not honored.

Storage engine characteristics are discussed in detail in the *MySQL Administrator's Guide*.

The other table options are used to optimize the behavior of the table. In most cases, you don't have to specify any of them. The options work for all storage engines unless otherwise indicated:

- AUTO_INCREMENT

  The initial AUTO_INCREMENT value for the table. This works for MyISAM only. To set the first auto-increment value for an InnoDB table, insert a dummy row with a value one less than the desired value after creating the table, and then delete the dummy row.

- `AVG_ROW_LENGTH`

  An approximation of the average row length for your table. You need to set this only for large tables with variable-size records.

  When you create a `MyISAM` table, MySQL uses the product of the `MAX_ROWS` and `AVG_ROW_LENGTH` options to decide how big the resulting table will be. If you don't specify either option, the maximum size for a table will be 4GB (or 2GB if your operating system only supports 2GB tables). The reason for this is just to keep down the pointer sizes to make the index smaller and faster if you don't really need big files. If you want all your tables to be able to grow above the 4GB limit and are willing to have your smaller tables slightly slower and larger than necessary, you may increase the default pointer size by setting the `myisam_data_pointer_size` system variable, which was added in MySQL 4.1.2.

- `CHECKSUM`

  Set this to 1 if you want MySQL to maintain a live checksum for all rows (that is, a checksum that MySQL updates automatically as the table changes). This makes the table a little slower to update, but also makes it easier to find corrupted tables. The `CHECKSUM TABLE` statement reports the checksum. (`MyISAM` only.)

- `COMMENT`

  A comment for your table, up to 60 characters long.

- `MAX_ROWS`

  The maximum number of rows you plan to store in the table.

- `MIN_ROWS`

  The minimum number of rows you plan to store in the table.

- `PACK_KEYS`

  Set this option to 1 if you want to have smaller indexes. This usually makes updates slower and reads faster. Setting the option to 0 disables all packing of keys. Setting it to `DEFAULT` (MySQL 4.0) tells the storage engine to only pack long `CHAR`/`VARCHAR` columns. (`MyISAM` and `ISAM` only.)

  If you don't use `PACK_KEYS`, the default is to only pack strings, not numbers. If you use `PACK_KEYS=1`, numbers will be packed as well.

  When packing binary number keys, MySQL uses prefix compression:

  - Every key needs one extra byte to indicate how many bytes of the previous key are the same for the next key.
  - The pointer to the row is stored in high-byte-first order directly after the key, to improve compression.

This means that if you have many equal keys on two consecutive rows, all following "same" keys will usually only take two bytes (including the pointer to the row). Compare this to the ordinary case where the following keys will take *storage_size_for_key* + *pointer_size* (where the pointer size is usually 4). Conversely, you will get a big benefit from prefix compression only if you have many numbers that are the same. If all keys are totally different, you will use one byte more per key, if the key isn't a key that can have NULL values. (In this case, the packed key length will be stored in the same byte that is used to mark if a key is NULL.)

- PASSWORD

  Encrypt the .frm file with a password. This option doesn't do anything in the standard MySQL version.

- DELAY_KEY_WRITE

  Set this to 1 if you want to delay key updates for the table until the table is closed. (MyISAM only.)

- ROW_FORMAT

  Defines how the rows should be stored. Currently, this option works only with MyISAM tables. The option value can be FIXED or DYNAMIC for static or variable-length row format. myisampack sets the type to COMPRESSED.

- RAID_TYPE

  The RAID_TYPE option can help you to exceed the 2GB/4GB limit for the MyISAM data file (not the index file) on operating systems that don't support big files. This option is unnecessary and not recommended for filesystems that support big files.

  You can get more speed from the I/O bottleneck by putting RAID directories on different physical disks. For now, the only allowed RAID_TYPE is STRIPED. 1 and RAID0 are aliases for STRIPED.

  If you specify the RAID_TYPE option for a MyISAM table, specify the RAID_CHUNKS and RAID_CHUNKSIZE options as well. The maximum RAID_CHUNKS value is 255. MyISAM will create RAID_CHUNKS subdirectories named 00, 01, 02, ... 09, 0a, 0b, ... in the database directory. In each of these directories, MyISAM will create a file tbl_name.MYD. When writing data to the data file, the RAID handler maps the first RAID_CHUNKSIZE*1024 bytes to the first file, the next RAID_CHUNKSIZE*1024 bytes to the next file, and so on.

  RAID_TYPE works on any operating system, as long as you have built MySQL with the --with-raid option to configure. To determine whether a server supports RAID tables, use SHOW VARIABLES LIKE 'have_raid' to see whether the variable value is YES.

- UNION

  UNION is used when you want to use a collection of identical tables as one. This works only with MERGE tables.

  For the moment, you must have SELECT, UPDATE, and DELETE privileges for the tables you map to a MERGE table. Originally, all used tables had to be in the same database as the MERGE table itself. This restriction has been lifted as of MySQL 4.1.1.

- INSERT_METHOD

  If you want to insert data in a MERGE table, you have to specify with INSERT_METHOD into which table the row should be inserted. INSERT_METHOD is an option useful for MERGE tables only. This option was introduced in MySQL 4.0.0.

- DATA DIRECTORY, INDEX DIRECTORY

  By using DATA DIRECTORY='*directory*' or INDEX DIRECTORY='*directory*' you can specify where the MyISAM storage engine should put a table's data file and index file. Note that the directory should be a full path to the directory (not a relative path).

  These options work only for MyISAM tables from MySQL 4.0 on, when you are not using the --skip-symbolic-links option. Your operating system must also have a working, thread-safe realpath() call.

As of MySQL 3.23, you can create one table from another by adding a SELECT statement at the end of the CREATE TABLE statement:

```
CREATE TABLE new_tbl SELECT * FROM orig_tbl;
```

MySQL will create new columns for all elements in the SELECT. For example:

```
mysql> CREATE TABLE test (a INT NOT NULL AUTO_INCREMENT,
    ->        PRIMARY KEY (a), KEY(b))
    ->        TYPE=MyISAM SELECT b,c FROM test2;
```

This creates a MyISAM table with three columns, a, b, and c. Notice that the columns from the SELECT statement are appended to the right side of the table, not overlapped onto it. Take the following example:

```
mysql> SELECT * FROM foo;
+---+
| n |
+---+
| 1 |
+---+

mysql> CREATE TABLE bar (m INT) SELECT n FROM foo;
Query OK, 1 row affected (0.02 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM bar;
+------+---+
| m    | n |
+------+---+
| NULL | 1 |
+------+---+
1 row in set (0.00 sec)
```

For each row in table `foo`, a row is inserted in `bar` with the values from `foo` and default values for the new columns.

If any errors occur while copying the data to the table, it is automatically dropped and not created.

`CREATE TABLE ... SELECT` will not automatically create any indexes for you. This is done intentionally to make the statement as flexible as possible. If you want to have indexes in the created table, you should specify these before the `SELECT` statement:

```
mysql> CREATE TABLE bar (UNIQUE (n)) SELECT n FROM foo;
```

Some conversion of column types might occur. For example, the `AUTO_INCREMENT` attribute is not preserved, and `VARCHAR` columns can become `CHAR` columns.

When creating a table with `CREATE ... SELECT`, make sure to alias any function calls or expressions in the query. If you do not, the `CREATE` statement might fail or result in undesirable column names.

```
CREATE TABLE artists_and_works
SELECT artist.name, COUNT(work.artist_id) AS number_of_works
FROM artist LEFT JOIN work ON artist.id = work.artist_id
GROUP BY artist.id;
```

As of MySQL 4.1, you can explicitly specify the type for a generated column:

```
CREATE TABLE foo (a TINYINT NOT NULL) SELECT b+1 AS a FROM bar;
```

In MySQL 4.1, you can also use `LIKE` to create an empty table based on the definition of another table, including any column attributes and indexes the original table has:

```
CREATE TABLE new_tbl LIKE orig_tbl;
```

`CREATE TABLE ... LIKE` does not copy any `DATA DIRECTORY` or `INDEX DIRECTORY` table options that were specified for the original table.

You can precede the `SELECT` by `IGNORE` or `REPLACE` to indicate how to handle records that duplicate unique key values. With `IGNORE`, new records that duplicate an existing record on a unique key value are discarded. With `REPLACE`, new records replace records that have the same unique key value. If neither `IGNORE` nor `REPLACE` is specified, duplicate unique key values result in an error.

To ensure that the update log/binary log can be used to re-create the original tables, MySQL will not allow concurrent inserts during `CREATE TABLE ... SELECT`.

### 6.2.5.1 Creating Foreign Keys

Starting from MySQL 3.23.44, InnoDB features foreign key constraints.

The syntax of a foreign key constraint definition in InnoDB looks like this:

```
[CONSTRAINT symbol] FOREIGN KEY [id] (index_col_name, ...)
    REFERENCES tbl_name (index_col_name, ...)
    [ON DELETE {CASCADE | SET NULL | NO ACTION | RESTRICT | SET DEFAULT}]
    [ON UPDATE {CASCADE | SET NULL | NO ACTION | RESTRICT | SET DEFAULT}]
```

Both tables must be InnoDB type. In the referencing table, there must be an index where the foreign key columns are listed as the *first* columns in the same order. In the referenced table, there must be an index where the referenced columns are listed as the *first* columns in the same order. Index prefix columns on foreign keys are not supported.

InnoDB needs indexes on foreign keys and referenced keys so that foreign key checks can be fast and not require a table scan. Starting with MySQL 4.1.2, these indexes are created automatically. In older versions, the indexes must be created explicitly or the creation of foreign key constraints will fail.

Corresponding columns in the foreign key and the referenced key must have similar internal data types inside InnoDB so that they can be compared without a type conversion. The **size and the signedness of integer types has to be the same**. The length of string types need not be the same. If you specify a SET NULL action, make sure you have **not declared the columns in the child table** as NOT NULL.

If MySQL reports an error number 1005 from a CREATE TABLE statement, and the error message string refers to errno 150, this means that the table creation failed because a foreign key constraint was not correctly formed. Similarly, if an ALTER TABLE fails and it refers to errno 150, that means a foreign key definition would be incorrectly formed for the altered table. Starting from MySQL 4.0.13, you can use SHOW INNODB STATUS to display a detailed explanation of the latest InnoDB foreign key error in the server.

Starting from MySQL 3.23.50, InnoDB does not check foreign key constraints on those foreign key or referenced key values that contain a NULL column.

**A deviation from SQL standards:** If in the parent table there are several rows that have the same referenced key value, then InnoDB acts in foreign key checks as if the other parent rows with the same key value do not exist. For example, if you have defined a RESTRICT type constraint, and there is a child row with several parent rows, InnoDB does not allow the deletion of any of those parent rows.

Starting from MySQL 3.23.50, you can also associate the ON DELETE CASCADE or ON DELETE SET NULL clause with the foreign key constraint. Corresponding ON UPDATE options are available starting from 4.0.8. If ON DELETE CASCADE is specified, and a row in the parent table is deleted, InnoDB automatically deletes also all those rows in the child table whose foreign key values are equal to the referenced key value in the parent row. If ON DELETE SET NULL is specified, the child rows are automatically updated so that the columns in the foreign key are set to the SQL NULL value. SET DEFAULT is parsed but ignored.

InnoDB performs cascading operations through a depth-first algorithm, based on records in the indexes corresponding to the foreign key constraints.

**A deviation from SQL standards:** If ON UPDATE CASCADE or ON UPDATE SET NULL recurses to update the *same table* it has already updated during the cascade, it acts like RESTRICT. This means that you cannot use self-referential ON UPDATE CASCADE or ON UPDATE SET NULL operations. This is to prevent infinite loops resulting from cascaded updates. A self-referential ON DELETE SET NULL, on the other hand, is possible from 4.0.13. A self-referential ON DELETE CASCADE has been possible since ON DELETE was implemented.

A simple example that relates parent and child tables through a single-column foreign key:

```
CREATE TABLE parent(id INT NOT NULL,
                    PRIMARY KEY (id)
) TYPE=INNODB;
CREATE TABLE child(id INT, parent_id INT,
                   INDEX par_ind (parent_id),
                   FOREIGN KEY (parent_id) REFERENCES parent(id)
                     ON DELETE CASCADE
) TYPE=INNODB;
```

A more complex example is one in which a product_order table has foreign keys for two other tables. One foreign key references a two-column index in the product table. The other references a single-column index in the customer table:

```
CREATE TABLE product (category INT NOT NULL, id INT NOT NULL,
                      price DECIMAL,
                      PRIMARY KEY(category, id)) TYPE=INNODB;
CREATE TABLE customer (id INT NOT NULL,
                      PRIMARY KEY (id)) TYPE=INNODB;
CREATE TABLE product_order (no INT NOT NULL AUTO_INCREMENT,
                      product_category INT NOT NULL,
                      product_id INT NOT NULL,
                      customer_id INT NOT NULL,
                      PRIMARY KEY(no),
                      INDEX (product_category, product_id),
                      FOREIGN KEY (product_category, product_id)
                        REFERENCES product(category, id)
                        ON UPDATE CASCADE ON DELETE RESTRICT,
                      INDEX (customer_id),
                      FOREIGN KEY (customer_id)
                        REFERENCES customer(id)) TYPE=INNODB;
```

Starting from MySQL 3.23.50, the InnoDB parser allows you to use backticks around table and column names in a FOREIGN KEY ... REFERENCES ... clause. Starting from MySQL 4.0.5, the InnoDB parser also takes into account the lower_case_table_names system variable setting.

### 6.2.5.2 Silent Column Specification Changes

In some cases, MySQL silently changes column specifications from those given in a `CREATE TABLE` or `ALTER TABLE` statement:

- `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 becomes 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.
- From MySQL 4.1.0 on, a `CHAR` or `VARCHAR` column with a length specification greater than 255 is converted to the smallest `TEXT` type that can hold values of the given length. For example, `VARCHAR(500)` is converted to `TEXT`, and `VARCHAR(200000)` is converted to `MEDIUMTEXT`. This is a compatibility feature.
- `TIMESTAMP` display sizes are discarded from MySQL 4.1 on, due to changes made to the `TIMESTAMP` column type in that version. Before MySQL 4.1, `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 can be assigned `NULL` values.
- Columns that are part of a `PRIMARY KEY` are made `NOT NULL` even if not declared that way.
- Starting from MySQL 3.23.51, trailing spaces are automatically deleted from `ENUM` and `SET` member values when the table is created.
- MySQL maps certain column types used by other SQL database vendors to MySQL types. See Section 4.7, "Using Column Types from Other Database Engines."
- If you include a `USING` clause to specify an index type that is not legal for a storage engine, but there is another index type available that the engine can use without affecting query results, the engine will use the available type.

To see whether MySQL used a column type other than the one you specified, issue a `DESCRIBE` or `SHOW CREATE TABLE` statement after creating or altering your table.

Certain other column type changes can occur if you compress a table using `myisampack`.

## 6.2.6 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 statement! To use `DROP DATABASE`, you need the `DROP` privilege on the database.

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.

If you use `DROP DATABASE` on a symbolically linked database, both the link and the original database are deleted.

As of MySQL 4.1.2, `DROP DATABASE` returns the number of tables that were removed. This corresponds to the number of `.frm` files removed.

The `DROP DATABASE` statement removes from the given database directory those files and directories that MySQL itself may create during normal operation:

- All files with these extensions:

|  |  |  |  |
|---|---|---|---|
| `.BAK` | `.DAT` | `.HSH` | `.ISD` |
| `.ISM` | `.ISM` | `.MRG` | `.MYD` |
| `.MYI` | `.db` | `.frm` |  |

- All subdirectories with names that consist of two hex digits `00-ff`. These are subdirectories used for `RAID` tables.
- The `db.opt` file, if it exists.

If other files or directories remain in the database directory after MySQL removes those just listed, the database directory cannot be removed. In this case, you must remove any remaining files or directories manually and issue the `DROP DATABASE` statement again.

You can also drop databases with `mysqladmin`.

## 6.2.7 DROP INDEX Syntax

```
DROP INDEX index_name ON tbl_name
```

`DROP INDEX` drops the index named *`index_name`* from the table *`tbl_name`*. In MySQL 3.22 or later, `DROP INDEX` is mapped to an `ALTER TABLE` statement to drop the index. See Section 6.2.2, "`ALTER TABLE` Syntax." `DROP INDEX` doesn't do anything prior to MySQL 3.22.

### 6.2.8 DROP TABLE Syntax

```
DROP [TEMPORARY] TABLE [IF EXISTS]
    tbl_name [, tbl_name] ...
    [RESTRICT | CASCADE]
```

`DROP TABLE` removes one or more tables. You must have the `DROP` privilege for each table. All table data and the table definition are *removed*, so *be careful* with this statement!

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. As of MySQL 4.1, a `NOTE` is generated for each non-existent table when using `IF EXISTS`. See Section 6.5.3.20, "`SHOW WARNINGS` Syntax."

`RESTRICT` and `CASCADE` are allowed to make porting easier. For the moment, they do nothing.

**Note:** `DROP TABLE` automatically commits the current active transaction, unless you are using MySQL 4.1 or higher and the `TEMPORARY` keyword.

The `TEMPORARY` keyword is ignored in MySQL 4.0. As of 4.1, it has the following effect:

- The statement drops only `TEMPORARY` tables.
- The statement doesn't end a running transaction.
- No access rights are checked. (A `TEMPORARY` table is visible only to the client that created it, so no check is necessary.)

Using `TEMPORARY` is a good way to ensure that you don't accidentally drop a non-`TEMPORARY` table.

### 6.2.9 RENAME TABLE Syntax

```
RENAME TABLE tbl_name TO new_tbl_name
    [, tbl_name2 TO new_tbl_name2] ...
```

This statement renames one or more tables. It was added in MySQL 3.23.23.

The rename operation is done atomically, which means that no other thread can access any of the tables while the rename is running. For example, if you have an existing table `old_table`, you can create another table `new_table` that has the same structure but is empty, and then replace the existing table with the empty one as follows:

```
CREATE TABLE new_table (...);
RENAME TABLE old_table TO backup_table, new_table TO old_table;
```

If the statement renames more than one table, renaming operations are done from left to right. If you want to swap two table names, you can do so like this (assuming that no table named `tmp_table` currently exists):

```
RENAME TABLE old_table TO tmp_table,
             new_table TO old_table,
             tmp_table TO new_table;
```

As long as two databases are on the same filesystem, you can also rename a table to move it from one database to another:

```
RENAME TABLE current_db.tbl_name TO other_db.tbl_name;
```

When you execute `RENAME`, you can't have any locked tables or active transactions. You must also have the `ALTER` and `DROP` privileges on the original table, and the `CREATE` and `INSERT` privileges on the new table.

If MySQL encounters any errors in a multiple-table rename, it will do a reverse rename for all renamed tables to get everything back to the original state.

## 6.3 MySQL Utility Statements

### 6.3.1 `DESCRIBE` Syntax (Get Information About Columns)

```
{DESCRIBE | DESC} tbl_name [col_name | wild]
```

`DESCRIBE` provides information about a table's columns. It is a shortcut for `SHOW COLUMNS FROM`.

See Section 6.5.3.4, "`SHOW COLUMNS` Syntax."

*`col_name`* can be a column name, or a string containing the SQL '`%`' and '`_`' wildcard characters to obtain output only for the columns with names matching the string. There is no need to enclose the string in quotes unless it contains spaces or other special characters.

If the column types are different from what you expect them to be based on a `CREATE TABLE` statement, note that MySQL sometimes changes column types. See Section 6.2.5.2, "Silent Column Specification Changes."

The `DESCRIBE` statement is provided for Oracle compatibility.

The `SHOW CREATE TABLE` and `SHOW TABLE STATUS` statements also provide information about tables. See Section 6.5.3, "`SET` and `SHOW` Syntax."

### 6.3.2 `USE` Syntax

```
USE db_name
```

The `USE` *`db_name`* statement tells MySQL to use the *`db_name`* database as the default (current) database for subsequent statements. The database remains the default 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 following example 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.

## 6.4 MySQL Transactional and Locking Statements

### 6.4.1 START TRANSACTION, COMMIT, and ROLLBACK Syntax

By default, MySQL runs with autocommit mode enabled. This means that as soon as you execute a statement that updates (modifies) a table, MySQL stores the update on disk.

If you are using transaction-safe tables (like InnoDB or BDB), you can disable autocommit mode with the following statement:

```
SET AUTOCOMMIT=0;
```

After disabling autocommit mode by setting the AUTOCOMMIT variable to zero, you must use COMMIT to store your changes to disk or ROLLBACK if you want to ignore the changes you have made since the beginning of your transaction.

If you want to disable autocommit mode for a single series of statements, you can use the START TRANSACTION statement:

```
START TRANSACTION;
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
UPDATE table2 SET summary=@A WHERE type=1;
COMMIT;
```

With START TRANSACTION, autocommit remains disabled until you end the transaction with COMMIT or ROLLBACK. The autocommit mode then reverts to its previous state.

BEGIN and BEGIN WORK can be used instead of START TRANSACTION to initiate a transaction. START TRANSACTION was added in MySQL 4.0.11. This is standard SQL syntax and is the recommended way to start an ad-hoc transaction. BEGIN and BEGIN WORK are available from MySQL 3.23.17 and 3.23.19, respectively.

Note that if you are not using transaction-safe tables, any changes are stored at once, regardless of the status of autocommit mode.

If you issue a ROLLBACK statement after updating a non-transactional table within a transaction, an ER_WARNING_NOT_COMPLETE_ROLLBACK warning occurs. Changes to transaction-safe tables will be rolled back, but not changes to non-transaction-safe tables.

If you are using `START TRANSACTION` or `SET AUTOCOMMIT=0`, you should use the MySQL binary log for backups instead of the older update log. Transactions are stored in the binary log in one chunk, upon `COMMIT`. Transactions that are rolled back are not logged. (Exception: Modifications to non-transactional tables cannot be rolled back. If a transaction that is rolled back includes modifications to non-transactional tables, the entire transaction is logged with a `ROLLBACK` statement at the end to ensure that the modifications to those tables are replicated. This is true as of MySQL 4.0.15.)

You can change the isolation level for transactions with `SET TRANSACTION ISOLATION LEVEL`. See Section 6.4.6, "`SET TRANSACTION` Syntax."

## 6.4.2 Statements That Cannot Be Rolled Back

Some statements cannot be rolled back. In general, these include data definition language (DDL) statements, such as those that create or drop databases, or those that create, drop, or alter tables.

You should design your transactions not to include such statements. If you issue a statement early in a transaction that cannot be rolled back, and then another statement later fails, the full effect of the transaction cannot be rolled back by issuing a `ROLLBACK` statement.

## 6.4.3 Statements That Cause an Implicit Commit

The following statements implicitly end a transaction (as if you had done a `COMMIT` before executing the statement):

| | | |
|---|---|---|
| `ALTER TABLE` | `BEGIN` | `CREATE INDEX` |
| `DROP DATABASE` | `DROP INDEX` | `DROP TABLE` |
| `LOAD MASTER DATA` | `LOCK TABLES` | `RENAME TABLE` |
| `SET AUTOCOMMIT=1` | `START TRANSACTION` | `TRUNCATE TABLE` |

`UNLOCK TABLES` also ends a transaction if any tables currently are locked. Prior to MySQL 4.0.13, `CREATE TABLE` ends a transaction if the binary update log is enabled.

Transactions cannot be nested. This is a consequence of the implicit `COMMIT` performed for any current transaction when you issue a `START TRANSACTION` statement or one of its synonyms.

## 6.4.4 `SAVEPOINT` and `ROLLBACK TO SAVEPOINT` Syntax

```
SAVEPOINT identifier
ROLLBACK TO SAVEPOINT identifier
```

Starting from MySQL 4.0.14 and 4.1.1, `InnoDB` supports the SQL statements `SAVEPOINT` and `ROLLBACK TO SAVEPOINT`.

The SAVEPOINT statement sets a named transaction savepoint with a name of *identifier*. If the current transaction already has a savepoint with the same name, the old savepoint is deleted and a new one is set.

The ROLLBACK TO SAVEPOINT statement rolls back a transaction to the named savepoint. Modifications that the current transaction made to rows after the savepoint was set are undone in the rollback, but InnoDB does *not* release the row locks that were stored in memory after the savepoint. (Note that for a new inserted row, the lock information is carried by the transaction ID stored in the row; the lock is not separately stored in memory. In this case, the row lock is released in the undo.) Savepoints that were set at a later time than the named savepoint are deleted.

If the statement returns the following error, it means that no savepoint with the specified name exists:

```
ERROR 1181: Got error 153 during ROLLBACK
```

All savepoints of the current transaction are deleted if you execute a COMMIT, or a ROLLBACK that does not name a savepoint.

## 6.4.5 LOCK TABLES and UNLOCK TABLES Syntax

```
LOCK TABLES
    tbl_name [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}
    [, tbl_name [AS alias] {READ [LOCAL] | [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 implicitly unlocked when the thread issues another LOCK TABLES, or when the connection to the server is closed.

**Note:** LOCK TABLES is not transaction-safe and implicitly commits any active transactions before attempting to lock the tables.

As of MySQL 4.0.2, to use LOCK TABLES you must have the LOCK TABLES privilege and a SELECT privilege for the involved tables. In MySQL 3.23, you must have SELECT, INSERT, DELETE, and UPDATE privileges for the tables.

The main reasons to use LOCK TABLES are for emulating transactions or to get more speed when updating tables. This is explained in more detail later.

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, only the thread holding the lock can read from or write to the table. Other threads are blocked.

The difference between READ LOCAL and READ is that READ LOCAL allows non-conflicting INSERT statements (concurrent inserts) to execute while the lock is held. However, this can't be used if you are going to manipulate the database files outside MySQL while you hold the lock.

When you use LOCK TABLES, you must lock all tables that you are going to use in your queries. If you are using a table multiple times in a query (with aliases), you must get a lock for each alias. While the locks obtained with a LOCK TABLES statement are in effect, you cannot access any tables that were not locked by the statement.

If your queries refer to a table using an alias, then you must lock the table using that same alias. It will not work to lock the table without specifying the alias:

```
mysql> LOCK TABLE t READ;
mysql> SELECT * FROM t AS myalias;
ERROR 1100: Table 'myalias' was not locked with LOCK TABLES
```

Conversely, if you lock a table using an alias, you must refer to it in your queries using that alias:

```
mysql> LOCK TABLE t AS myalias READ;
mysql> SELECT * FROM t;
ERROR 1100: Table 't' was not locked with LOCK TABLES
mysql> SELECT * FROM t AS myalias;
```

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 use LOW_PRIORITY WRITE locks only if you are sure that there will eventually be a time when no threads will have a READ lock.

LOCK TABLES works as follows:

1. Sort all tables to be locked in an internally defined order. From the user standpoint, this order is undefined.
2. If a table is locked with a read and a write lock, put the write lock before the read lock.
3. Lock one table at a time until the thread gets all locks.

This policy ensures that table locking is deadlock free. There are, however, other things you need to be aware of about this policy:

If you are using a LOW_PRIORITY WRITE lock for a table, it means only that MySQL will wait for this particular lock until there are no threads that want a READ lock. When the thread has gotten the WRITE lock and is waiting to get the lock for the next table in the lock table list, all other threads will wait for the WRITE lock to be released. If this becomes a serious problem with your application, you should consider converting some of your tables to transaction-safe tables.

You can safely use KILL to terminate a thread that is waiting for a table lock. See Section 6.5.4.3, “KILL Syntax.”

Note that you should *not* lock any tables that you are using with `INSERT DELAYED` because in that case the `INSERT` is done by a separate thread.

Normally, you don't have to lock tables, because 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 set of `MyISAM` tables, it's much faster to lock the tables you are going to use. Locking `MyISAM` tables speeds up inserting, updating, or deleting on them. The downside is that no thread can update a `READ`-locked table (including the one holding the lock) and no thread can access a `WRITE`-locked table other than the one holding the lock.

  The reason some `MyISAM` operations are faster under `LOCK TABLES` is that MySQL will not flush the key cache for the locked tables until `UNLOCK TABLES` is called. Normally, the key cache is flushed after each SQL statement.

- If you are using a storage engine in MySQL that doesn't support transactions, you must use `LOCK TABLES` if you want to ensure that no other thread comes between a `SELECT` and an `UPDATE`. The example shown here requires `LOCK TABLES` 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`, it is possible that another thread might insert a new row in the `trans` table between execution of the `SELECT` and `UPDATE` statements.

You can avoid using `LOCK TABLES` in many cases by using relative updates (`UPDATE customer SET value=value+new_value`) or the `LAST_INSERT_ID()` function, See Section 1.8.5.3, "Transactions and Atomic Operations."

You can also avoid locking tables in some cases by using the user-level advisory 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 5.8.4, "Miscellaneous Functions."

You can lock all tables in all databases with read locks with the `FLUSH TABLES WITH READ LOCK` statement. See Section 6.5.4.2, "`FLUSH` Syntax." This is a very convenient way to get backups if you have a filesystem such as Veritas that can take snapshots in time.

**Note:** If you use `ALTER TABLE` on a locked table, it may become unlocked. See Section A.3.1, "Problems with `ALTER TABLE`."

### 6.4.6 SET TRANSACTION Syntax

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }
```

This statement sets the transaction isolation level for the next transaction, globally, or for the current session.

The default behavior of `SET TRANSACTION` is to set the isolation level for the next (not yet started) transaction. If you use the `GLOBAL` keyword, the statement sets the default transaction level globally for all new connections created from that point on. Existing connections are unaffected. You need the `SUPER` privilege to do this. Using the `SESSION` keyword sets the default transaction level for all future transactions performed on the current connection.

For descriptions of each `InnoDB` transaction isolation level, see the *MySQL Administrator's Guide*. `InnoDB` supports each of these levels from MySQL 4.0.5 on. The default level is `REPEATABLE READ`.

You can set the initial default global isolation level for `mysqld` with the `--transaction-isolation` option.

## 6.5 Database Administration Statements

### 6.5.1 Account Management Statements

#### 6.5.1.1 DROP USER Syntax

```
DROP USER user
```

The `DROP USER` statement deletes a MySQL account that doesn't have any privileges. It serves to remove the account record from the `user` table. The account is named using the same format as for `GRANT` or `REVOKE`; for example, `'jeffrey'@'localhost'`. The user and host parts of the account name correspond to the `User` and `Host` column values of the `user` table record for the account.

To remove a MySQL user account, you should use the following procedure, performing the steps in the order shown:

1. Use `SHOW GRANTS` to determine what privileges the account has. See Section 6.5.3.10, "`SHOW GRANTS` Syntax."
2. Use `REVOKE` to revoke the privileges displayed by `SHOW GRANTS`. This removes records for the account from all the grant tables except the `user` table, and revokes any global privileges listed in the `user` table. See Section 6.5.1.2, "`GRANT` and `REVOKE` Syntax."
3. Delete the account by using `DROP USER` to remove the `user` table record.

The `DROP USER` statement was added in MySQL 4.1.1. Before 4.1.1, you should first revoke the account privileges as just described. Then delete the `user` table record and flush the grant tables like this:

```
mysql> DELETE FROM mysql.user
    -> WHERE User='user_name' and Host='host_name';
mysql> FLUSH PRIVILEGES;
```

### 6.5.1.2 GRANT and REVOKE Syntax

```
GRANT priv_type [(column_list)] [, priv_type [(column_list)]] ...
    ON {tbl_name | * | *.* | db_name.*}
    TO user [IDENTIFIED BY [PASSWORD] 'password']
        [, user [IDENTIFIED BY [PASSWORD] 'password']] ...
    [REQUIRE
        NONE |
        [{SSL| X509}]
        [CIPHER 'cipher' [AND]]
        [ISSUER 'issuer' [AND]]
        [SUBJECT 'subject']]
    [WITH [GRANT OPTION | MAX_QUERIES_PER_HOUR count |
                          MAX_UPDATES_PER_HOUR count |
                          MAX_CONNECTIONS_PER_HOUR count]]

REVOKE priv_type [(column_list)] [, priv_type [(column_list)]] ...
    ON {tbl_name | * | *.* | db_name.*}
    FROM user [, user] ...

REVOKE ALL PRIVILEGES, GRANT OPTION FROM user [, user] ...
```

The `GRANT` and `REVOKE` statements allow system administrators to create MySQL user accounts and to grant rights to and revoke them from accounts. `GRANT` and `REVOKE` are implemented in MySQL 3.22.11 or later. For earlier MySQL versions, these statements do nothing.

MySQL account information is stored in the tables of the `mysql` database. This database and the access control system are discussed extensively in the *MySQL Administrator's Guide*, which you should consult for additional details.

Privileges can be granted at four levels:

- **Global level**

  Global privileges apply to all databases on a given server. These privileges are stored in the `mysql.user` table. `GRANT ALL ON *.*` and `REVOKE ALL ON *.*` grant and revoke only global privileges.

- **Database level**

  Database privileges apply to all tables in a given database. These privileges are stored in the `mysql.db` and `mysql.host` tables. `GRANT ALL ON` *`db_name`*`.*` and `REVOKE ALL ON` *`db_name`*`.*` grant and revoke only database privileges.
- **Table level**

  Table privileges apply to all columns in a given table. These privileges are stored in the `mysql.tables_priv` table. `GRANT ALL ON` *`db_name.tbl_name`* and `REVOKE ALL ON` *`db_name.tbl_name`* grant and revoke only table privileges.
- **Column level**

  Column privileges apply to single columns in a given table. These privileges are stored in the `mysql.columns_priv` table. When using `REVOKE`, you must specify the same columns that were granted.

To make it easy to revoke all privileges, MySQL 4.1.2 has added the following syntax, which drops all database-, table-, and column-level privileges for the named users:

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM user [, user] ...
```

Before MySQL 4.1.2, all privileges cannot be dropped at once. Two statements are necessary:

```
REVOKE ALL PRIVILEGES FROM user [, user] ...
REVOKE GRANT OPTION FROM user [, user] ...
```

For the `GRANT` and `REVOKE` statements, *`priv_type`* can be specified as any of the following:

| Privilege | Meaning |
|---|---|
| `ALL [PRIVILEGES]` | Sets all simple privileges except `GRANT OPTION` |
| `ALTER` | Allows use of `ALTER TABLE` |
| `CREATE` | Allows use of `CREATE TABLE` |
| `CREATE TEMPORARY TABLES` | Allows use of `CREATE TEMPORARY TABLE` |
| `DELETE` | Allows use of `DELETE` |
| `DROP` | Allows use of `DROP TABLE` |
| `EXECUTE` | Allows the user to run stored procedures (MySQL 5.0) |
| `FILE` | Allows use of `SELECT ... INTO OUTFILE` and `LOAD DATA INFILE` |
| `INDEX` | Allows use of `CREATE INDEX` and `DROP INDEX` |
| `INSERT` | Allows use of `INSERT` |
| `LOCK TABLES` | Allows use of `LOCK TABLES` on tables for which you have the `SELECT` privilege |
| `PROCESS` | Allows use of `SHOW FULL PROCESSLIST` |
| `REFERENCES` | Not yet implemented |

| Privilege | Meaning |
|---|---|
| `RELOAD` | Allows use of `FLUSH` |
| `REPLICATION CLIENT` | Allows the user to ask where the slave or master servers are |
| `REPLICATION SLAVE` | Needed for replication slaves (to read binary log events from the master) |
| `SELECT` | Allows use of `SELECT` |
| `SHOW DATABASES` | `SHOW DATABASES` shows all databases |
| `SHUTDOWN` | Allows use of `mysqladmin shutdown` |
| `SUPER` | Allows use of `CHANGE MASTER`, `KILL`, `PURGE MASTER LOGS`, and `SET GLOBAL` statements, the `mysqladmin debug command`; allows you to connect (once) even if `max_connections` is reached |
| `UPDATE` | Allows use of `UPDATE` |
| `USAGE` | Synonym for "no privileges" |
| `GRANT OPTION` | Allows privileges to be granted |

`USAGE` can be used when you want to create a user that has no privileges.

The privileges `CREATE TEMPORARY TABLES`, `EXECUTE`, `LOCK TABLES`, `REPLICATION ...`, `SHOW DATABASES`, and `SUPER` are new in MySQL 4.0.2. To use these new privileges after upgrading to 4.0.2, you must run the `mysql_fix_privilege_tables` script.

In older MySQL versions that do not have the `SUPER` privilege, the `PROCESS` privilege can be used instead.

You can assign global privileges by using `ON *.*` syntax or database privileges by using `ON` *`db_name`*`.*` syntax. If you specify `ON *` and you have a current database, the privileges will be granted in that database. (**Warning:** If you specify `ON *` and you *don't* have a current database, the privileges granted will be global!)

The `EXECUTION`, `FILE`, `PROCESS`, `RELOAD`, `REPLICATION CLIENT`, `REPLICATION SLAVE`, `SHOW DATABASES`, `SHUTDOWN`, and `SUPER` privileges are administrative privileges that can only be granted globally (using `ON *.*` syntax).

Other privileges can be granted globally or at more specific levels.

The only *`priv_type`* values you can specify for a table are `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `DROP`, `GRANT OPTION`, `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`.

`GRANT ALL` assigns only the privileges that exist at the level you are granting. For example, if you use `GRANT ALL ON` *`db_name`*`.*`, that is a database-level statement, so none of the global-only privileges such as `FILE` will be granted.

MySQL allows you to create database-level privileges even if the database doesn't exist, to make it easy to prepare for database use. However, MySQL currently does not allow you to create table-level privileges if the table doesn't exist.

MySQL does not automatically revoke any privileges even if you drop a table or drop a database.

**Note:** the '_' and '%' wildcards are allowed when specifying database names in `GRANT` statements that grant privileges at the global or database levels. This means, for example, that if you want to use a '_' character as part of a database name, you should specify it as '_' in the `GRANT` statement, to prevent the user from being able to access additional databases matching the wildcard pattern; for example, ``GRANT ... ON `foo\_bar`.* TO ....``

In order to accommodate granting rights to users from arbitrary hosts, MySQL supports specifying the *`user`* value in the form *`user_name@host_name`*. If you want to specify a *`user_name`* string containing special characters (such as '-'), or a *`host_name`* string containing special characters or wildcard characters (such as '%'), you can quote the username or hostname (for example, `'test-user'@'test-hostname'`). Quote the username and hostname separately.

You can specify wildcards in the hostname. For example, *`user_name`*`@'%.loc.gov'` applies to *`user_name`* for any host in the `loc.gov` domain, and *`user_name`*`@'144.155.166.%'` applies to *`user_name`* for any host in the `144.155.166` class C subnet.

The simple form *`user_name`* is a synonym for *`user_name`*`@'%'`.

MySQL doesn't support wildcards in usernames. Anonymous users are defined by inserting entries with `User=''` into the `mysql.user` table or creating a user with an empty name with the `GRANT` statement:

```
mysql> GRANT ALL ON test.* TO ''@'localhost' ...
```

**Warning:** If you allow anonymous users to connect to the MySQL server, you should also grant privileges to all local users as *`user_name`*`@localhost`. Otherwise, the anonymous-user account for the local host in the `mysql.user` table will be used when named users try to log in to the MySQL server from the local machine! (This anonymous-user account is created during MySQL installation.)

You can determine whether this applies to you by executing the following query:

```
mysql> SELECT Host, User FROM mysql.user WHERE User='';
```

If you want to delete the local anonymous-user account to avoid the problem just described, use these statements:

```
mysql> DELETE FROM mysql.user WHERE Host='localhost' AND User='';
mysql> FLUSH PRIVILEGES;
```

For the moment, `GRANT` only supports host, table, database, and column names up to 60 characters long. A username can be up to 16 characters.

The privileges for a table or column are formed additively 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, the privilege cannot 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 this complicated.

If you grant privileges for a username/hostname combination that does not exist in the `mysql.user` table, an entry is added and remains there until deleted with a `DELETE` statement. In other words, `GRANT` may create `user` table entries, but `REVOKE` will not remove them; you must do that explicitly using `DROP USER` or `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 is 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` statement. See Section 6.5.1.3, "`SET PASSWORD` Syntax."

If you don't want to send the password in clear text, you can use the `PASSWORD` keyword followed by a scrambled password from the `PASSWORD()` SQL function or the `make_scrambled_password()` C API function.

If you grant privileges for a database, an entry in the `mysql.db` table is created if needed. If all privileges for the database are removed with `REVOKE`, this entry is deleted.

If a user has no privileges for a table, the table name is not displayed when the user requests a list of tables (for example, with a `SHOW TABLES` statement). If a user has no privileges for a database, the database name is not displayed by `SHOW DATABASES` unless the user has the `SHOW DATABASES` privilege.

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 OPTION` privilege, because 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 OPTION` privilege allows you to give away only those privileges you possess.

Be aware that when you grant a user the `GRANT OPTION` 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 that 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 `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!

The `MAX_QUERIES_PER_HOUR` *`count`*, `MAX_UPDATES_PER_HOUR` *`count`*, and `MAX_CONNECTIONS_PER_HOUR` *`count`* options are new in MySQL 4.0.2. They limit the number of queries, updates, and logins a user can perform during one hour. If *`count`* is 0 (the default), this means there is no limitation for that user. Note: To specify any of these options for an existing user without affecting existing privileges, use `GRANT USAGE ON *.* ... WITH MAX_...`.

MySQL can check X509 certificate attributes in addition to the usual authentication that is based on the username and password. To specify SSL-related options for a MySQL account, use the `REQUIRE` clause of the `GRANT` statement.

There are different possibilities for limiting connection types for an account:

- If an account has no SSL or X509 requirements, unencrypted connections are allowed if the username and password are valid. However, encrypted connections also can be used at the client's option, if the client has the proper certificate and key files.
- The `REQUIRE SSL` option tells the server to allow only SSL-encrypted connections for the account. Note that this option can be omitted if there are any access-control records that allow non-SSL connections.

  ```
  mysql> GRANT ALL PRIVILEGES ON test.* TO 'root'@'localhost'
      -> IDENTIFIED BY 'goodsecret' REQUIRE SSL;
  ```

- `REQUIRE X509` means that the client must have a valid certificate but that the exact certificate, issuer, and subject do not matter. The only requirement is that it should be possible to verify its signature with one of the CA certificates.

  ```
  mysql> GRANT ALL PRIVILEGES ON test.* TO 'root'@'localhost'
      -> IDENTIFIED BY 'goodsecret' REQUIRE X509;
  ```

- `REQUIRE ISSUER 'issuer'` places the restriction on connection attempts that the client must present a valid X509 certificate issued by CA `'issuer'`. If the client presents a certificate that is valid but has a different issuer, the server rejects the connection. Use of X509 certificates always implies encryption, so the `SSL` option is unnecessary.

  ```
  mysql> GRANT ALL PRIVILEGES ON test.* TO 'root'@'localhost'
      -> IDENTIFIED BY 'goodsecret'
      -> REQUIRE ISSUER '/C=FI/ST=Some-State/L=Helsinki/
         O=MySQL Finland AB/CN=Tonu Samuel/Email=tonu@example.com';
  ```

  Note that the `ISSUER` value should be entered as a single string.

- `REQUIRE SUBJECT 'subject'` places the restriction on connection attempts that the client must present a valid X509 certificate with subject `'subject'` in it. If the client presents a certificate that is valid but has a different subject, the server rejects the connection.

  ```
  mysql> GRANT ALL PRIVILEGES ON test.* TO 'root'@'localhost'
      -> IDENTIFIED BY 'goodsecret'
      -> REQUIRE SUBJECT '/C=EE/ST=Some-State/L=Tallinn/
         O=MySQL demo client certificate/
         CN=Tonu Samuel/Email=tonu@example.com';
  ```

  Note that the `SUBJECT` value should be entered as a single string.

- `REQUIRE CIPHER 'cipher'` is needed to ensure that strong enough ciphers and key lengths will be used. SSL itself can be weak if old algorithms with short encryption keys are used. Using this option, you can ask for some exact cipher method to allow a connection.

  ```
  mysql> GRANT ALL PRIVILEGES ON test.* TO 'root'@'localhost'
      -> IDENTIFIED BY 'goodsecret'
      -> REQUIRE CIPHER 'EDH-RSA-DES-CBC3-SHA';
  ```

The `SUBJECT`, `ISSUER`, and `CIPHER` options can be combined in the `REQUIRE` clause like this:

```
mysql> GRANT ALL PRIVILEGES ON test.* TO 'root'@'localhost'
    -> IDENTIFIED BY 'goodsecret'
    -> REQUIRE SUBJECT '/C=EE/ST=Some-State/L=Tallinn/
       O=MySQL demo client certificate/
       CN=Tonu Samuel/Email=tonu@example.com'
    -> AND ISSUER '/C=FI/ST=Some-State/L=Helsinki/
       O=MySQL Finland AB/CN=Tonu Samuel/Email=tonu@example.com'
    -> AND CIPHER 'EDH-RSA-DES-CBC3-SHA';
```

Note that the `SUBJECT` and `ISSUER` values each should be entered as a single string.

Starting from MySQL 4.0.4, the `AND` keyword is optional between `REQUIRE` options.

The order of the options does not matter, but no option can be specified twice.

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`, and so on), you should execute a `FLUSH PRIVILEGES` statement or run `mysqladmin flush-privileges` to tell the server to reload the grant 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 slows down MySQL a bit. Similarly, if you limit the number of queries, updates, or connections for any users, the server must monitor these values.

The biggest differences between the standard SQL and MySQL versions of `GRANT` are:

- In MySQL, privileges are associated with a username/hostname combination and not with only a username.
- Standard SQL doesn't have global or database-level privileges, nor does it support all the privilege types that MySQL supports.
- MySQL doesn't support the standard SQL `TRIGGER` or `UNDER` privileges.
- Standard SQL privileges are structured in a hierarchical manner. If you remove a user, all privileges the user has been granted are revoked. In MySQL, the granted privileges are not automatically revoked; you must revoke them yourself.
- With standard SQL, when you drop a table, all privileges for the table are revoked. With standard SQL, when you revoke a privilege, all privileges that were granted based on the privilege are also revoked. In MySQL, privileges can be dropped only with explicit `REVOKE` statements or by manipulating the MySQL grant tables.
- In MySQL, if you have the `INSERT` privilege on only some of the columns in a table, you can execute `INSERT` statements on the table; the columns for which you don't have the `INSERT` privilege will be set to their default values. Standard SQL requires you to have the `INSERT` privilege on all columns.

### 6.5.1.3 `SET PASSWORD` Syntax

```
SET PASSWORD = PASSWORD('some password')
SET PASSWORD FOR user = PASSWORD('some password')
```

The `SET PASSWORD` statement assigns a password to an existing MySQL user account.

The first syntax sets the password for the current user. Any client that has connected to the server using a non-anonymous account can change the password for that account.

The second syntax sets the password for a specific account on the current server host. Only clients with access to the `mysql` database can do this. The *`user`* value should be given in *`user_name@host_name`* format, where *`user_name`* and *`host_name`* 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` column values of `'bob'` and `'%.loc.gov'`, you would write the statement like this:

```
mysql> SET PASSWORD FOR 'bob'@'%.loc.gov' = PASSWORD('newpass');
```

That is equivalent to the following statements:

```
mysql> UPDATE mysql.user SET Password=PASSWORD('newpass')
    -> WHERE User='bob' AND Host='%.loc.gov';
mysql> FLUSH PRIVILEGES;
```

### 6.5.2 Table Maintenance Statements

#### 6.5.2.1 ANALYZE TABLE Syntax

```
ANALYZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name [, tbl_name] ...
```

This statement analyzes and stores the key distribution for a table. During the analysis, the table is locked with a read lock. This works on MyISAM and BDB tables and (as of MySQL 4.0.13) InnoDB tables. For MyISAM tables, this statement is equivalent to using `myisamchk -a`.

MySQL uses the stored key distribution to decide the order in which tables should be joined when you perform a join on something other than a constant.

The statement returns a table with the following columns:

| Column | Value |
|---|---|
| `Table` | The table name |
| `Op` | Always `analyze` |
| `Msg_type` | One of `status`, `error`, `info`, or `warning` |
| `Msg_text` | The message |

You can check the stored key distribution with the SHOW INDEX statement. See Section 6.5.3.7, "SHOW DATABASES Syntax."

If the table hasn't changed since the last ANALYZE TABLE statement, the table will not be analyzed again.

Before MySQL 4.1.1, ANALYZE TABLE statements are not written to the binary log. As of MySQL 4.1.1, they are written to the binary log unless the optional NO_WRITE_TO_BINLOG keyword (or its alias LOCAL) is used.

#### 6.5.2.2 BACKUP TABLE Syntax

```
BACKUP TABLE tbl_name [, tbl_name] ... TO '/path/to/backup/directory'
```

**Note:** This statement is deprecated. We are working on a better replacement for it that will provide online backup capabilities. In the meantime, the `mysqlhotcopy` script can be used instead.

BACKUP TABLE copies to the backup directory the minimum number of table files needed to restore the table, after flushing any buffered changes to disk. The statement works only for MyISAM tables. It copies the `.frm` definition and `.MYD` data files. The `.MYI` index file can be rebuilt from those two files. The directory should be specified as a full pathname.

During the backup, a read lock is held for each table, one at time, as they are being backed up. If you want to back up several tables as a snapshot (preventing any of them from being changed during the backup operation), you must first issue a `LOCK TABLES` statement to obtain a read lock for every table in the group.

The statement returns a table with the following columns:

| Column | Value |
|---|---|
| `Table` | The table name |
| `Op` | Always `backup` |
| `Msg_type` | One of `status`, `error`, `info`, or `warning` |
| `Msg_text` | The message |

`BACKUP TABLE` is available in MySQL 3.23.25 and later.

#### 6.5.2.3 `CHECK TABLE` Syntax

```
CHECK TABLE tbl_name [, tbl_name] ... [option] ...

option = {QUICK | FAST | MEDIUM | EXTENDED | CHANGED}
```

`CHECK TABLE` works only on `MyISAM` and `InnoDB` tables. On `MyISAM` tables, it's the same thing as running `myisamchk --medium-check` *`tbl_name`* on the table.

If you don't specify any option, `MEDIUM` is used.

Checks the table or tables for errors. For `MyISAM` tables, the key statistics are updated. The statement returns a table with the following columns:

| Column | Value |
|---|---|
| `Table` | The table name |
| `Op` | Always `check` |
| `Msg_type` | One of `status`, `error`, `info`, or `warning` |
| `Msg_text` | The message |

Note that the statement might produce many rows of information for each checked table. The last row will have a `Msg_type` value of `status` and the `Msg_text` normally should be `OK`. If you don't get `OK`, or `Table is already up to date`, you should normally run a repair of the table. `Table is already up to date` means that the storage engine for the table indicated that there was no need to check the table.

The different check types are as follows:

| Type | Meaning |
|---|---|
| `QUICK` | Don't scan the rows to check for incorrect links. |
| `FAST` | Only check tables that haven't been closed properly. |
| `CHANGED` | Only check tables that have been changed since the last check or haven't been closed properly. |
| `MEDIUM` | Scan rows to verify that deleted links are okay. This also calculates a key checksum for the rows and verifies this with a calculated checksum for the keys. |
| `EXTENDED` | Do a full key lookup for all keys for each row. This ensures that the table is 100% consistent, but will take a long time! |

If none of the options `QUICK`, `MEDIUM`, or `EXTENDED` are specified, the default check type for dynamic-format `MyISAM` tables is `MEDIUM`. The default check type also is `MEDIUM` for static-format `MyISAM` tables, unless `CHANGED` or `FAST` is specified. In that case, the default is `QUICK`. The row scan is skipped for `CHANGED` and `FAST` because the rows are very seldom corrupted.

You can combine check options, as in the following example, which does a quick check on the table to see whether it was closed properly:

```
CHECK TABLE test_table FAST QUICK;
```

**Note:** In some cases, `CHECK TABLE` will change the table! This happens if the table is marked as "corrupted" or "not closed properly" but `CHECK TABLE` doesn't find any problems in the table. In this case, `CHECK TABLE` marks the table as okay.

If a table is corrupted, it's most likely that the problem is in the indexes and not in the data part. All of the preceding check types check the indexes thoroughly and should thus find most errors.

If you just want to check a table that you assume is okay, you should use no check options or the `QUICK` option. The latter should be used when you are in a hurry and can take the very small risk that `QUICK` doesn't find an error in the data file. (In most cases, MySQL should find, under normal usage, any error in the data file. If this happens, the table is marked as "corrupted" and cannot be used until it's repaired.)

`FAST` and `CHANGED` are mostly intended to be used from a script (for example, to be executed from `cron`) if you want to check your table from time to time. In most cases, `FAST` is to be preferred over `CHANGED`. (The only case when it isn't preferred is when you suspect that you have found a bug in the `MyISAM` code.)

`EXTENDED` is to be used only after you have run a normal check but still get strange errors from a table when MySQL tries to update a row or find a row by key. (This is very unlikely if a normal check has succeeded!)

Some problems reported by `CHECK TABLE` can't be corrected automatically:

- `Found row where the auto_increment column has the value 0.`

  This means that you have a row in the table where the `AUTO_INCREMENT` index column contains the value 0. (It's possible to create a row where the `AUTO_INCREMENT` column is 0 by explicitly setting the column to 0 with an `UPDATE` statement.)

  This isn't an error in itself, but could cause trouble if you decide to dump the table and restore it or do an `ALTER TABLE` on the table. In this case, the `AUTO_INCREMENT` column will change value according to the rules of `AUTO_INCREMENT` columns, which could cause problems such as a duplicate-key error.

  To get rid of the warning, just execute an `UPDATE` statement to set the column to some other value than 0.

### 6.5.2.4 `CHECKSUM TABLE` Syntax

```
CHECKSUM TABLE tbl_name [, tbl_name] ... [ QUICK | EXTENDED ]
```

Reports a table checksum.

If `QUICK` is specified, the live table checksum is reported if it is available, or `NULL` otherwise. This is very fast. A live checksum is enabled by specifying the `CHECKSUM=1` table option, currently supported only for `MyISAM` tables. See Section 6.2.5, "`CREATE TABLE` Syntax."

In `EXTENDED` mode the whole table is read row by row and the checksum is calculated. This can be very slow for large tables.

By default, if neither `QUICK` nor `EXTENDED` is specified, MySQL returns a live checksum if the table storage engine supports it and scans the table otherwise.

This statement is implemented in MySQL 4.1.1.

### 6.5.2.5 `OPTIMIZE TABLE` Syntax

```
OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name [, 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 and to defragment the data file.

In most setups, you need not run `OPTIMIZE TABLE` at all. Even if you do a lot of updates to variable-length rows, it's not likely that you need to do this more than once a week or month and only on certain tables.

For the moment, `OPTIMIZE TABLE` works only on `MyISAM` and `BDB` tables. For `BDB` tables, `OPTIMIZE TABLE` is currently mapped to `ANALYZE TABLE`. See Section 6.5.2.1, "`ANALYZE TABLE` Syntax."

You can get OPTIMIZE TABLE to work on other table types by starting mysqld with the --skip-new or --safe-mode option, but in this case, OPTIMIZE TABLE is just mapped to ALTER TABLE.

OPTIMIZE TABLE works as follows:

1. If the table has deleted or split rows, repair the table.
2. If the index pages are not sorted, sort them.
3. If the statistics are not up to date (and the repair couldn't be done by sorting the index), update them.

Note that MySQL locks the table during the time OPTIMIZE TABLE is running.

Before MySQL 4.1.1, OPTIMIZE TABLE statements are not written to the binary log. As of MySQL 4.1.1, they are written to the binary log unless the optional NO_WRITE_TO_BINLOG keyword (or its alias LOCAL) is used.

#### 6.5.2.6 REPAIR TABLE Syntax

```
REPAIR [LOCAL | NO_WRITE_TO_BINLOG] TABLE
    tbl_name [, tbl_name] ... [QUICK] [EXTENDED] [USE_FRM]
```

REPAIR TABLE repairs a possibly corrupted table. By default, it has the same effect as myisamchk --recover *tbl_name*. REPAIR TABLE works only on MyISAM tables.

Normally, you should never have to run this statement. However, if disaster strikes, REPAIR TABLE is very likely to get back all your data from a MyISAM table. If your tables become corrupted often, you should try to find the reason for it, to eliminate the need to use REPAIR TABLE.

The statement returns a table with the following columns:

| Column | Value |
|---|---|
| Table | The table name |
| Op | Always repair |
| Msg_type | One of status, error, info, or warning |
| Msg_text | The message |

The REPAIR TABLE statement might produce many rows of information for each repaired table. The last row will have a Msg_type value of status and Msg_test normally should be OK. If you don't get OK, you should try repairing the table with myisamchk --safe-recover, because REPAIR TABLE does not yet implement all the options of myisamchk. We plan to make it more flexible in the future.

If QUICK is given, REPAIR TABLE tries to repair only the index tree. This type of repair is like that done by myisamchk --recover --quick.

If you use `EXTENDED`, MySQL creates the index row by row instead of creating one index at a time with sorting. (Before MySQL 4.1, this might be better than sorting on fixed-length keys if you have long `CHAR` keys that compress very well.) This type of repair is like that done by `myisamchk --safe-recover`.

As of MySQL 4.0.2, there is a `USE_FRM` mode for `REPAIR TABLE`. Use it if the `.MYI` index file is missing or if its header is corrupted. In this mode, MySQL will re-create the `.MYI` file using information from the `.frm` file. This kind of repair cannot be done with `myisamchk`.

**Warning:** If the server dies during a `REPAIR TABLE` operation, it's essential after restarting it that you immediately execute another `REPAIR TABLE` statement for the table before performing any other operations on it. (It's always good to start by making a backup.) In the worst case, you might have a new clean index file without information about the data file, and then the next operation you perform could overwrite the data file. This is an unlikely, but possible scenario.

Before MySQL 4.1.1, `REPAIR TABLE` statements are not written to the binary log. As of MySQL 4.1.1, they are written to the binary log unless the optional `NO_WRITE_TO_BINLOG` keyword (or its alias `LOCAL`) is used.

### 6.5.2.7 RESTORE TABLE Syntax

```
RESTORE TABLE tbl_name [, tbl_name] ... FROM '/path/to/backup/directory'
```

Restores the table or tables from a backup that was made with `BACKUP TABLE`. Existing tables will not be overwritten; if you try to restore over an existing table, you will get an error. Just as `BACKUP TABLE`, `RESTORE TABLE` currently works only for `MyISAM` tables. The directory should be specified as a full pathname.

The backup for each table consists of its `.frm` format file and `.MYD` data file. The restore operation restores those files, then uses them to rebuild the `.MYI` index file. Restoring takes longer than backing up due to the need to rebuild the indexes. The more indexes the table has, the longer it will take.

The statement returns a table with the following columns:

| Column | Value |
|---|---|
| `Table` | The table name |
| `Op` | Always `restore` |
| `Msg_type` | One of `status`, `error`, `info`, or `warning` |
| `Msg_text` | The message |

### 6.5.3 `SET` and `SHOW` Syntax

`SET` allows you to set variables and options.

`SHOW` has many forms that provide information about databases, tables, columns, or status information about the server. This section describes those following:

```
SHOW [FULL] COLUMNS FROM tbl_name [FROM db_name] [LIKE 'pattern']
SHOW CREATE DATABASE db_name
SHOW CREATE TABLE tbl_name
SHOW DATABASES [LIKE 'pattern']
SHOW [STORAGE] ENGINES
SHOW ERRORS [LIMIT [offset,] row_count]
SHOW GRANTS FOR user
SHOW INDEX FROM tbl_name [FROM db_name]
SHOW INNODB STATUS
SHOW [BDB] LOGS
SHOW PRIVILEGES
SHOW [FULL] PROCESSLIST
SHOW STATUS [LIKE 'pattern']
SHOW TABLE STATUS [FROM db_name] [LIKE 'pattern']
SHOW [OPEN] TABLES [FROM db_name] [LIKE 'pattern']
SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'pattern']
SHOW WARNINGS [LIMIT [offset,] row_count]
```

If the syntax for a given `SHOW` statement includes a `LIKE 'pattern'` part, `'pattern'` is a string that can contain the SQL '`%`' and '_' wildcard characters. The pattern is useful for restricting statement output to matching values.

Note that there are other forms of these statements described elsewhere:

- The `SET PASSWORD` statement for assigning account passwords is described in Section 6.5.1.3, "`SET PASSWORD` Syntax."
- The `SHOW` statement has forms that provide information about replication master and slave servers:

  ```
  SHOW BINLOG EVENTS
  SHOW MASTER LOGS
  SHOW MASTER STATUS
  SHOW SLAVE HOSTS
  SHOW SLAVE STATUS
  ```

  These forms of `SHOW` are described in Section 6.6, "Replication Statements."

### 6.5.3.1 SET Syntax

```
SET variable_assignment [, variable_assignment] ...

variable_assignment:
      user_var_name = expr
    | [GLOBAL | SESSION] system_var_name = expr
    | @@[global. | session.]system_var_name = expr
```

`SET` sets different types of variables that affect the operation of the server or your client. It can be used to assign values to user variables or system variables.

In MySQL 4.0.3, we added the `GLOBAL` and `SESSION` options and allowed most important system variables to be changed dynamically at runtime.

In older versions of MySQL, `SET OPTION` is used instead of `SET`, but this is now deprecated; just leave out the word `OPTION`.

The following examples show the different syntaxes you can use to set variables.

A user variable is written as `@`*`var_name`* and can be set as follows:

```
SET @var_name = expr;
```

Further information about user variables is given in Section 2.3, "User Variables."

System variables can be referred to in `SET` statements as *`var_name`*. The name optionally can be preceded by `GLOBAL` or `@@global.` to indicate explicitly that the variable is a global variable, or by `SESSION`, `@@session.`, or `@@` to indicate that it is a session variable. `LOCAL` and `@@local.` are synonyms for `SESSION` and `@@session.`. If no modifier is present, `SET` sets the session variable.

The `@@`*`var_name`* syntax for system variables is supported to make MySQL syntax compatible with some other database systems.

If you set several system variables in the same statement, the last used `GLOBAL` or `SESSION` option is used for variables that have no mode specified.

```
SET sort_buffer_size=10000;
SET @@local.sort_buffer_size=10000;
SET GLOBAL sort_buffer_size=1000000, SESSION sort_buffer_size=1000000;
SET @@sort_buffer_size=1000000;
SET @@global.sort_buffer_size=1000000, @@local.sort_buffer_size=1000000;
```

If you set a system variable using `SESSION` (the default), the value remains in effect until the current session ends or until you set the variable to a different value. If you set a system variable using `GLOBAL`, which requires the `SUPER` privilege, the value is remembered and used for new connections until the server restarts. If you want to make a variable setting permanent, you should put it in an option file.

To prevent incorrect usage, MySQL produces an error if you use `SET GLOBAL` with a variable that can only be used with `SET SESSION` or if you do not specify `GLOBAL` when setting a global variable.

If you want to set a `SESSION` variable to the `GLOBAL` value or a `GLOBAL` value to the compiled-in MySQL default value, you can set it to `DEFAULT`. For example, the following two statements are identical in setting the session value of `max_join_size` to the global value:

```
SET max_join_size=DEFAULT;
SET @@session.max_join_size=@@global.max_join_size;
```

You can get a list of most system variables with `SHOW VARIABLES`. See Section 6.5.3.19, "`SHOW VARIABLES` Syntax." To get a specific variable name or list of names that match a pattern, use a `LIKE` clause:

```
SHOW VARIABLES LIKE 'max_join_size';
SHOW GLOBAL VARIABLES LIKE 'max_join_size';
```

You can also get the value for a specific value by using the `@@[global.|local.]`*`var_name`* syntax with `SELECT`:

```
SELECT @@max_join_size, @@global.max_join_size;
```

When you retrieve a variable with `SELECT @@`*`var_name`* (that is, you do not specify `global.`, `session.`, or `local.`), MySQL returns the `SESSION` value if it exists and the `GLOBAL` value otherwise.

The following list describes variables that have non-standard syntax or that are not described in the list of system variables that is found in the *MySQL Administrator's Guide*. Although these variables are not displayed by `SHOW VARIABLES`, you can obtain their values with `SELECT` (with the exception of `CHARACTER SET` and `NAMES`). For example:

```
mysql> SELECT @@AUTOCOMMIT;
+--------------+
| @@autocommit |
+--------------+
|            1 |
+--------------+
```

- `AUTOCOMMIT = {0 | 1}`

  Set the autocommit mode. If set to `1`, all changes to a table take effect immediately. If set to `0`, you have to use `COMMIT` to accept a transaction or `ROLLBACK` to cancel it. If you change `AUTOCOMMIT` mode from `0` to `1`, MySQL performs an automatic `COMMIT` of any open transaction. Another way to begin a transaction is to use a `START TRANSACTION` or `BEGIN` statement. See Section 6.4.1, "`START TRANSACTION`, `COMMIT`, and `ROLLBACK` Syntax."

- `BIG_TABLES = {0 | 1}`

  If set to `1`, all temporary tables are stored on disk rather than in memory. This is a little slower, but the error `The table` *`tbl_name`* `is full` will not occur for `SELECT` operations that require a large temporary table. The default value for a new connection is `0` (use in-memory temporary tables). As of MySQL 4.0, you should normally never need to set this variable, because MySQL automatically converts in-memory tables to disk-based tables as necessary. This variable previously was named `SQL_BIG_TABLES`.

- `CHARACTER SET {`*`charset_name`*` | DEFAULT}`

  This maps all strings from and to the client with the given mapping. Before MySQL 4.1, the only allowable value for *`charset_name`* is `cp1251_koi8`, but you can add new mappings by editing the `sql/convert.cc` file in the MySQL source distribution. As of MySQL 4.1.1, `SET CHARACTER SET` sets three session system variables: `character_set_client` and `character_set_results` are set to the given character set, and `character_set_connection` to the value of `character_set_database`.

  The default mapping can be restored by using a value of `DEFAULT`.

  Note that the syntax for `SET CHARACTER SET` differs from that for setting most other options.

- `FOREIGN_KEY_CHECKS = {0 | 1}`

  If set to `1` (the default), foreign key constraints for `InnoDB` tables are checked. If set to `0`, they are ignored. Disabling foreign key checking can be useful for reloading `InnoDB` tables in an order different than that required by their parent/child relationships. This variable was added in MySQL 3.23.52.

- `IDENTITY = `*`value`*

  The variable is a synonym for the `LAST_INSERT_ID` variable. It exists for compatibility with other databases. As of MySQL 3.23.25, you can read its value with `SELECT @@IDENTITY`. As of MySQL 4.0.3, you can also set its value with `SET IDENTITY`.

- `INSERT_ID = `*`value`*

  Set the value to be used by the following `INSERT` or `ALTER TABLE` statement when inserting an `AUTO_INCREMENT` value. This is mainly used with the binary log.

- `LAST_INSERT_ID = `*`value`*

  Set the value to be returned from `LAST_INSERT_ID()`. This is stored in the binary log when you use `LAST_INSERT_ID()` in a statement that updates a table. Setting this variable does not update the value returned by the `mysql_insert_id()` C API function.

- `NAMES {'`*`charset_name`*`' | DEFAULT}`

  `SET NAMES` sets the three session system variables `character_set_client`, `character_set_connection`, and `character_set_results` to the given character set.

  The default mapping can be restored by using a value of `DEFAULT`.

  Note that the syntax for `SET NAMES` differs from that for setting most other options. This statement is available as of MySQL 4.1.0.

- `SQL_AUTO_IS_NULL = {0 | 1}`

  If set to `1` (the default), you can find the last inserted row for a table that contains an `AUTO_INCREMENT` column by using the following construct:

  ```
  WHERE auto_increment_column IS NULL
  ```

  This behavior is used by some ODBC programs, such as Access. `SQL_AUTO_IS_NULL` was added in MySQL 3.23.52.

- `SQL_BIG_SELECTS = {0 | 1}`

  If set to `0`, MySQL aborts `SELECT` statements that probably will take a very long time (that is, statements for which the optimizer estimates that the number of examined rows will exceed the value of `max_join_size`). This is useful when an inadvisable `WHERE` statement has been issued. The default value for a new connection is `1`, which allows all `SELECT` statements.

  If you set the `max_join_size` system variable to a value other than `DEFAULT`, `SQL_BIG_SELECTS` will be set to `0`.

- `SQL_BUFFER_RESULT = {0 | 1}`

  `SQL_BUFFER_RESULT` forces results from `SELECT` statements to be put into temporary tables. This helps MySQL free the table locks early and can be beneficial in cases where it takes a long time to send results to the client. This variable was added in MySQL 3.23.13.

- `SQL_LOG_BIN = {0 | 1}`

  If set to `0`, no logging is done to the binary log for the client. The client must have the `SUPER` privilege to set this option. This variable was added in MySQL 3.23.16.

- `SQL_LOG_OFF = {0 | 1}`

  If set to `1`, no logging is done to the general query log for this client. The client must have the `SUPER` privilege to set this option.

- `SQL_LOG_UPDATE = {0 | 1}`

  If set to `0`, no logging is done to the update log for the client. The client must have the `SUPER` privilege to set this option. This variable was added in MySQL 3.22.5. Starting from MySQL 5.0.0, it is deprecated and is mapped to `SQL_LOG_BIN`.

- `SQL_QUOTE_SHOW_CREATE = {0 | 1}`

  If set to `1`, `SHOW CREATE TABLE` quotes table and column names. If set to `0`, quoting is disabled. This option is enabled by default so that replication will work for tables with table and column names that require quoting. This variable was added in MySQL 3.23.26. Section 6.5.3.6, "`SHOW CREATE TABLE` Syntax."

- `SQL_SAFE_UPDATES = {0 | 1}`

  If set to `1`, MySQL aborts `UPDATE` or `DELETE` statements that do not use a key in the `WHERE` clause or a `LIMIT` clause. This makes it possible to catch `UPDATE` or `DELETE` statements where keys are not used properly and that would probably change or delete a large number of rows. This variable was added in MySQL 3.22.32.

- `SQL_SELECT_LIMIT = {`*`value`*` | DEFAULT}`

  The maximum number of records to return from `SELECT` statements. 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`.

  If a `SELECT` has a `LIMIT` clause, the `LIMIT` takes precedence over the value of `SQL_SELECT_LIMIT`.

- `SQL_WARNINGS = {0 | 1}`

  This variable controls whether single-row `INSERT` statements produce an information string if warnings occur. The default is 0. Set the value to 1 to produce an information string. This variable was added in MySQL 3.22.11.

- `TIMESTAMP = {`*`timestamp_value`*` | DEFAULT}`

  Set the time for this client. This is used to get the original timestamp if you use the binary log to restore rows. *`timestamp_value`* should be a Unix epoch timestamp, not a MySQL timestamp.

- `UNIQUE_CHECKS = {0 | 1}`

  If set to `1` (the default), uniqueness checks for secondary indexes in `InnoDB` tables are performed. If set to `0`, no uniqueness checks are done. This variable was added in MySQL 3.23.52.

## 6.5.3.2 SHOW CHARACTER SET Syntax

```
SHOW CHARACTER SET [LIKE 'pattern']
```

The `SHOW CHARACTER SET` statement shows all available character sets. It takes an optional `LIKE` clause that indicates which character set names to match. For example:

```
mysql> SHOW CHARACTER SET LIKE 'latin%';
+---------+-----------------------------+-------------------+--------+
| Charset | Description                 | Default collation | Maxlen |
+---------+-----------------------------+-------------------+--------+
| latin1  | ISO 8859-1 West European    | latin1_swedish_ci |      1 |
| latin2  | ISO 8859-2 Central European | latin2_general_ci |      1 |
| latin5  | ISO 8859-9 Turkish          | latin5_turkish_ci |      1 |
| latin7  | ISO 8859-13 Baltic          | latin7_general_ci |      1 |
+---------+-----------------------------+-------------------+--------+
```

The `Maxlen` column shows the maximum number of bytes used to store one character.

`SHOW CHARACTER SET` is available as of MySQL 4.1.0.

### 6.5.3.3 SHOW COLLATION Syntax

```
SHOW COLLATION [LIKE 'pattern']
```

The output from `SHOW COLLATION` includes all available character sets. It takes an optional `LIKE` clause that indicates which collation names to match. For example:

```
mysql> SHOW COLLATION LIKE 'latin1%';
+-------------------+---------+----+---------+----------+---------+
| Collation         | Charset | Id | Default | Compiled | Sortlen |
+-------------------+---------+----+---------+----------+---------+
| latin1_german1_ci | latin1  |  5 |         |          |       0 |
| latin1_swedish_ci | latin1  |  8 | Yes     | Yes      |       0 |
| latin1_danish_ci  | latin1  | 15 |         |          |       0 |
| latin1_german2_ci | latin1  | 31 |         | Yes      |       2 |
| latin1_bin        | latin1  | 47 |         | Yes      |       0 |
| latin1_general_ci | latin1  | 48 |         |          |       0 |
| latin1_general_cs | latin1  | 49 |         |          |       0 |
| latin1_spanish_ci | latin1  | 94 |         |          |       0 |
+-------------------+---------+----+---------+----------+---------+
```

The `Default` column indicates whether a collation is the default for its character set. `Compiled` indicates whether the character set is compiled into the server. `Sortlen` is related to the amount of memory required to sort strings expressed in the character set.

`SHOW COLLATION` is available as of MySQL 4.1.0.

### 6.5.3.4 SHOW COLUMNS Syntax

```
SHOW [FULL] COLUMNS FROM tbl_name [FROM db_name] [LIKE 'pattern']
```

`SHOW COLUMNS` lists the columns in a given table. If the column types differ from what you expect them to be based on your `CREATE TABLE` statement, note that MySQL sometimes changes column types when you create or alter a table. The conditions for which this occurs are described in Section 6.2.5.2, "Silent Column Specification Changes."

The `FULL` keyword can be used from MySQL 3.23.32 on. It causes the output to include the privileges you have for each column. As of MySQL 4.1, `FULL` also causes any per-column comments to be displayed.

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

`SHOW FIELDS` is a synonym for `SHOW COLUMNS`. You can also list a table's columns with the `mysqlshow` *`db_name tbl_name`* command.

The `DESCRIBE` statement provides information similar to `SHOW COLUMNS`. See Section 6.3.1, "`DESCRIBE` Syntax (Get Information About Columns)."

### 6.5.3.5 SHOW CREATE DATABASE Syntax

```
SHOW CREATE DATABASE db_name
```

Shows a CREATE DATABASE statement that will create the given database. It was added in MySQL 4.1.

```
mysql> SHOW CREATE DATABASE test\G
*************************** 1. row ***************************
       Database: test
Create Database: CREATE DATABASE `test`
                 /*!40100 DEFAULT CHARACTER SET latin1 */
```

### 6.5.3.6 SHOW CREATE TABLE Syntax

```
SHOW CREATE TABLE tbl_name
```

Shows a CREATE TABLE statement that will create the given table. It was added in MySQL 3.23.20.

```
mysql> SHOW CREATE TABLE t\G
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE t (
  id INT(11) default NULL auto_increment,
  s char(60) default NULL,
  PRIMARY KEY (id)
) TYPE=MyISAM
```

SHOW CREATE TABLE quotes table and column names according to the value of the SQL_QUOTE_SHOW_CREATE option. Section 6.5.3.1, "SET Syntax."

### 6.5.3.7 SHOW DATABASES Syntax

```
SHOW DATABASES [LIKE 'pattern']
```

SHOW DATABASES lists the databases on the MySQL server host. You can also get this list using the mysqlshow command. As of MySQL 4.0.2, you will see only those databases for which you have some kind of privilege, if you don't have the global SHOW DATABASES privilege.

If the server was started with the --skip-show-database option, you cannot use this statement at all unless you have the SHOW DATABASES privilege.

### 6.5.3.8 `SHOW ENGINES` Syntax

```
SHOW [STORAGE] ENGINES
```

`SHOW ENGINES` shows you status information about the storage engines. This is particularly useful for checking whether a storage engine is supported, or to see what the default engine is. This statement is implemented in MySQL 4.1.2. `SHOW TABLE TYPES` is a deprecated synonym.

```
mysql> SHOW ENGINES\G
*************************** 1. row ***************************
   Type: MyISAM
Support: DEFAULT
Comment: Default type from 3.23 with great performance
*************************** 2. row ***************************
   Type: HEAP
Support: YES
Comment: Hash based, stored in memory, useful for temporary tables
*************************** 3. row ***************************
   Type: MEMORY
Support: YES
Comment: Alias for HEAP
*************************** 4. row ***************************
   Type: MERGE
Support: YES
Comment: Collection of identical MyISAM tables
*************************** 5. row ***************************
   Type: MRG_MYISAM
Support: YES
Comment: Alias for MERGE
*************************** 6. row ***************************
   Type: ISAM
Support: NO
Comment: Obsolete table type; Is replaced by MyISAM
*************************** 7. row ***************************
   Type: MRG_ISAM
Support: NO
Comment: Obsolete table type; Is replaced by MRG_MYISAM
*************************** 8. row ***************************
   Type: InnoDB
Support: YES
Comment: Supports transactions, row-level locking and foreign keys
*************************** 9. row ***************************
   Type: INNOBASE
Support: YES
Comment: Alias for INNODB
```

```
*************************** 10. row ***************************
   Type: BDB
Support: YES
Comment: Supports transactions and page-level locking
*************************** 11. row ***************************
   Type: BERKELEYDB
Support: YES
Comment: Alias for BDB
```

A `Support` value indicates whether the particular storage engine is supported, and which is the default engine. For example, if the server is started with the `--default-table-type=InnoDB` option, then the `Support` value for the `InnoDB` row will have the value `DEFAULT`.

### 6.5.3.9 `SHOW ERRORS` Syntax

```
SHOW ERRORS [LIMIT [offset,] row_count]
SHOW COUNT(*) ERRORS
```

This statement is similar to `SHOW WARNINGS`, except that instead of displaying errors, warnings, and notes, it displays only errors. `SHOW ERRORS` is available as of MySQL 4.1.0.

The `LIMIT` clause has the same syntax as for the `SELECT` statement. See Section 6.1.7, "`SELECT` Syntax."

The `SHOW COUNT(*) ERRORS` statement displays the number of errors. You can also retrieve this number from the `error_count` variable:

```
SHOW COUNT(*) ERRORS;
SELECT @@error_count;
```

For more information, see Section 6.5.3.20, "`SHOW WARNINGS` Syntax."

### 6.5.3.10 `SHOW GRANTS` Syntax

```
SHOW GRANTS FOR user
```

This statement lists the `GRANT` statements that must be issued to duplicate the privileges for a MySQL user account.

```
mysql> SHOW GRANTS FOR 'root'@'localhost';
+---------------------------------------------------------------------+
| Grants for root@localhost                                           |
+---------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION |
+---------------------------------------------------------------------+
```

As of MySQL 4.1.2, to list privileges for the current session, you can use any of the following statements:

```
SHOW GRANTS;
SHOW GRANTS FOR CURRENT_USER;
SHOW GRANTS FOR CURRENT_USER();
```

Before MySQL 4.1.2, you can find out what user the session was authenticated as by selecting the value of the `CURRENT_USER()` function (new in MySQL 4.0.6). Then use that value in the `SHOW GRANTS` statement. See Section 5.8.3, "Information Functions."

`SHOW GRANTS` is available as of MySQL 3.23.4.

#### 6.5.3.11 `SHOW INDEX` Syntax

```
SHOW INDEX FROM tbl_name [FROM db_name]
```

`SHOW INDEX` returns table index information in a format that resembles the `SQLStatistics` call in ODBC.

`SHOW INDEX` returns the following fields:

- `Table`

  The name of the table.

- `Non_unique`

  0 if the index can't contain duplicates, 1 if it can.

- `Key_name`

  The name of the index.

- `Seq_in_index`

  The column sequence number in the index, starting with 1.

- `Column_name`

  The column name.

- `Collation`

  How the column is sorted in the index. In MySQL, this can have values '`A`' (Ascending) or `NULL` (Not sorted).

- `Cardinality`

  The number of unique values in the index. This is updated by running `ANALYZE TABLE` or `myisamchk -a`. `Cardinality` is counted based on statistics stored as integers, so it's not necessarily accurate for small tables.

- `Sub_part`

  The number of indexed characters if the column is only partly indexed. `NULL` if the entire column is indexed.

- `Packed`

  Indicates how the key is packed. `NULL` if it is not.
- `Null`

  Contains `YES` if the column may contain `NULL`, `''` if not.
- `Index_type`

  The index method used (`BTREE`, `FULLTEXT`, `HASH`, `RTREE`).
- `Comment`

  Various remarks. Before MySQL 4.0.2 when the `Index_type` column was added, `Comment` indicates whether an index is `FULLTEXT`.

The `Packed` and `Comment` columns were added in MySQL 3.23.0. The `Null` and `Index_type` columns were added in MySQL 4.0.2.

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 KEYS` is a synonym for `SHOW INDEX`. You can also list a table's indexes with the `mysqlshow -k` *`db_name tbl_name`* command.

### 6.5.3.12 `SHOW INNODB STATUS` Syntax

```
SHOW INNODB STATUS
```

This statement shows extensive information about the state of the `InnoDB` storage engine.

### 6.5.3.13 `SHOW LOGS` Syntax

```
SHOW [BDB] LOGS
```

`SHOW LOGS` displays status information about existing log files. It was implemented in MySQL 3.23.29. Currently, it displays only information about Berkeley DB log files, so an alias for it (available as of MySQL 4.1.1) is `SHOW BDB LOGS`.

`SHOW LOGS` returns the following fields:

- `File`

  The full path to the log file.
- `Type`

  The log file type (`BDB` for Berkeley DB log files).
- `Status`

  The status of the log file (`FREE` if the file can be removed, or `IN USE` if the file is needed by the transaction subsystem).

### 6.5.3.14 SHOW PRIVILEGES Syntax

```
SHOW PRIVILEGES
```

SHOW PRIVILEGES shows the list of system privileges that the underlying MySQL server supports. This statement is implemented as of MySQL 4.1.0.

```
mysql> SHOW PRIVILEGES\G
*************************** 1. row ***************************
Privilege: Select
  Context: Tables
  Comment: To retrieve rows from table
*************************** 2. row ***************************
Privilege: Insert
  Context: Tables
  Comment: To insert data into tables
*************************** 3. row ***************************
Privilege: Update
  Context: Tables
  Comment: To update existing rows
*************************** 4. row ***************************
Privilege: Delete
  Context: Tables
  Comment: To delete existing rows
*************************** 5. row ***************************
Privilege: Index
  Context: Tables
  Comment: To create or drop indexes
*************************** 6. row ***************************
Privilege: Alter
  Context: Tables
  Comment: To alter the table
*************************** 7. row ***************************
Privilege: Create
  Context: Databases,Tables,Indexes
  Comment: To create new databases and tables
*************************** 8. row ***************************
Privilege: Drop
  Context: Databases,Tables
  Comment: To drop databases and tables
*************************** 9. row ***************************
Privilege: Grant
  Context: Databases,Tables
  Comment: To give to other users those privileges you possess
*************************** 10. row ***************************
Privilege: References
  Context: Databases,Tables
  Comment: To have references on tables
```

```
*************************** 11. row ***************************
Privilege: Reload
  Context: Server Admin
  Comment: To reload or refresh tables, logs and privileges
*************************** 12. row ***************************
Privilege: Shutdown
  Context: Server Admin
  Comment: To shutdown the server
*************************** 13. row ***************************
Privilege: Process
  Context: Server Admin
  Comment: To view the plain text of currently executing queries
*************************** 14. row ***************************
Privilege: File
  Context: File access on server
  Comment: To read and write files on the server
```

### 6.5.3.15 SHOW PROCESSLIST Syntax

```
SHOW [FULL] PROCESSLIST
```

`SHOW PROCESSLIST` shows you which threads are running. You can also get this information using the `mysqladmin processlist` statement. If you have the `SUPER` privilege, you can see all threads. Otherwise, you can see only your own threads (that is, threads associated with the MySQL account that you are using). See Section 6.5.4.3, "`KILL` Syntax." If you don't use the `FULL` keyword, only the first 100 characters of each query are shown.

Starting from MySQL 4.0.12, the statement reports the hostname for TCP/IP connections in *`host_name`*:*`client_port`* format to make it easier to determine which client is doing what.

This statement is very useful if you get the "too many connections" error message and want to find out what is going on. MySQL reserves one extra connection to be used by accounts that have the `SUPER` privilege, to ensure that administrators should always be able to connect and check the system (assuming that you are not giving this privilege to all your users).

Some states commonly seen in the output from `SHOW PROCESSLIST`:

- `Checking table`

  The thread is performing (automatic) checking of the table.
- `Closing tables`

  Means that the thread is flushing the changed table data to disk and closing the used tables. This should be a fast operation. If not, then you should verify that you don't have a full disk and that the disk is not in very heavy use.
- `Connect Out`

  Slave connecting to master.

- `Copying to tmp table on disk`

  The temporary result set was larger than `tmp_table_size` and the thread is now changing the temporary table from in-memory to disk-based format to save memory.
- `Creating tmp table`

  The thread is creating a temporary table to hold a part of the result for the query.
- `deleting from main table`

  The server is executing the first part of a multiple-table delete and deleting only from the first table.
- `deleting from reference tables`

  The server is executing the second part of a multiple-table delete and deleting the matched rows from the other tables.
- `Flushing tables`

  The thread is executing `FLUSH TABLES` and is waiting for all threads to close their tables.
- `Killed`

  Someone has sent a kill to the thread and it should abort next time it checks the kill flag. The flag is checked in each major loop in MySQL, but in some cases it might still take a short time for the thread to die. If the thread is locked by some other thread, the kill takes effect as soon as the other thread releases its lock.
- `Sending data`

  The thread is processing rows for a `SELECT` statement and also is sending data to the client.
- `Sorting for group`

  The thread is doing a sort to satisfy a `GROUP BY`.
- `Sorting for order`

  The thread is doing a sort to satisfy an `ORDER BY`.
- `Opening tables`

  The thread is trying to open a table. This is should be a very fast procedure, unless something prevents opening. For example, an `ALTER TABLE` or a `LOCK TABLE` statement can prevent opening a table until the statement is finished.
- `Removing duplicates`

  The query was using `SELECT DISTINCT` in such a way that MySQL couldn't optimize away the distinct operation at an early stage. Because of this, MySQL requires an extra stage to remove all duplicated rows before sending the result to the client.
- `Reopen table`

  The thread got a lock for the table, but noticed after getting the lock that the underlying table structure changed. It has freed the lock, closed the table, and is now trying to reopen it.

- `Repair by sorting`

  The repair code is using sorting to create indexes.

- `Repair with keycache`

  The repair code is using creating keys one by one through the key cache. This is much slower than `Repair by sorting`.

- `Searching rows for update`

  The thread is doing a first phase to find all matching rows before updating them. This has to be done if the `UPDATE` is changing the index that is used to find the involved rows.

- `Sleeping`

  The thread is waiting for the client to send a new statement to it.

- `System lock`

  The thread is waiting to get an external system lock for the table. If you are not using multiple `mysqld` servers that are accessing the same tables, you can disable system locks with the `--skip-external-locking` option.

- `Upgrading lock`

  The `INSERT DELAYED` handler is trying to get a lock for the table to insert rows.

- `Updating`

  The thread is searching for rows to update and updating them.

- `User Lock`

  The thread is waiting on a `GET_LOCK()`.

- `Waiting for tables`

  The thread got a notification that the underlying structure for a table has changed and it needs to reopen the table to get the new structure. However, to be able to reopen the table, it must wait until all other threads have closed the table in question.

  This notification happens if another thread has used `FLUSH TABLES` or one of the following statements on the table in question: `FLUSH TABLES` *`tbl_name`*, `ALTER TABLE`, `RENAME TABLE`, `REPAIR TABLE`, `ANALYZE TABLE`, or `OPTIMIZE TABLE`.

- `waiting for handler insert`

  The `INSERT DELAYED` handler has processed all pending inserts and is waiting for new ones.

Most states correspond to very quick operations. If a thread stays in any of these states for many seconds, there might be a problem that needs to be investigated.

There are some other states that are not mentioned in the preceding list, but many of them are useful only for finding bugs in the server.

### 6.5.3.16 SHOW STATUS Syntax

```
SHOW STATUS [LIKE 'pattern']
```

SHOW STATUS provides server status information. This information also can be obtained using the mysqladmin extended-status command.

Partial output is shown here. The list of variables and their values may be different for your server. The meaning of each variable is given in the *MySQL Administrator's Guide*.

```
mysql> SHOW STATUS;
+--------------------------+------------+
| Variable_name            | Value      |
+--------------------------+------------+
| Aborted_clients          | 0          |
| Aborted_connects         | 0          |
| Bytes_received           | 155372598  |
| Bytes_sent               | 1176560426 |
| Connections              | 30023      |
| Created_tmp_disk_tables  | 0          |
| Created_tmp_tables       | 8340       |
| Created_tmp_files        | 60         |
...
| Open_tables              | 1          |
| Open_files               | 2          |
| Open_streams             | 0          |
| Opened_tables            | 44600      |
| Questions                | 2026873    |
...
| Table_locks_immediate    | 1920382    |
| Table_locks_waited       | 0          |
| Threads_cached           | 0          |
| Threads_created          | 30022      |
| Threads_connected        | 1          |
| Threads_running          | 1          |
| Uptime                   | 80380      |
+--------------------------+------------+
```

With a LIKE clause, the statement displays only those variables that match the pattern:

```
mysql> SHOW STATUS LIKE 'Key%';
+--------------------+----------+
| Variable_name      | Value    |
+--------------------+----------+
| Key_blocks_used    | 14955    |
| Key_read_requests  | 96854827 |
| Key_reads          | 162040   |
| Key_write_requests | 7589728  |
| Key_writes         | 3813196  |
+--------------------+----------+
```

### 6.5.3.17 `SHOW TABLE STATUS` Syntax

```
SHOW TABLE STATUS [FROM db_name] [LIKE 'pattern']
```

`SHOW TABLE STATUS` (new in MySQL 3.23) works likes `SHOW TABLE`, but provides a lot of information about each table. You can also get this list using the `mysqlshow --status` *`db_name`* command.

`SHOW TABLE STATUS` returns the following fields:

- `Name`

  The name of the table.
- `Type`

  The type of the table.
- `Row_format`

  The row storage format (`Fixed`, `Dynamic`, or `Compressed`).
- `Rows`

  The number of rows.
- `Avg_row_length`

  The average row length.
- `Data_length`

  The length of the data file.
- `Max_data_length`

  The maximum length of the data file. For fixed-row formats, this is the maximum number of rows in the table. For dynamic-row formats, this is the total number of data bytes that can be stored in the table, given the data pointer size used.
- `Index_length`

  The length of the index file.
- `Data_free`

  The number of allocated but unused bytes.
- `Auto_increment`

  The next `AUTO_INCREMENT` value.
- `Create_time`

  When the table was created.
- `Update_time`

  When the data file was last updated.
- `Check_time`

  When the table was last checked.

- `Collation`

  The table's character set and collation. (New in 4.1.1)
- `Checksum`

  The live checksum value (if any). (New in 4.1.1)
- `Create_options`

  Extra options used with `CREATE TABLE`.
- `Comment`

  The comment used when creating the table (or some information why MySQL couldn't access the table information).

In the table comment, `InnoDB` tables will report the free space of the tablespace to which the table belongs. For a table located in the shared tablespace, this is the free space of the shared tablespace. If you are using multiple tablespaces and the table has its own tablespace, the freespace is for just that table.

For `MEMORY` (`HEAP`) tables, the `Data_length`, `Max_data_length`, and `Index_length` values approximate the actual amount of allocated memory. The allocation algorithm reserves memory in large amounts to reduce the number of allocation operations.

### 6.5.3.18 `SHOW TABLES` Syntax

```
SHOW [OPEN] TABLES [FROM db_name] [LIKE 'pattern']
```

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

**Note:** If you have no privileges for a table, the table will not show up in the output from `SHOW TABLES` or `mysqlshow` *`db_name`*.

`SHOW OPEN TABLES` lists the tables that are currently open in the table cache. The `Comment` field in the output tells how many times the table is `cached` and `in_use`. `OPEN` can be used from MySQL 3.23.33 on.

### 6.5.3.19 `SHOW VARIABLES` Syntax

```
SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'pattern']
```

`SHOW VARIABLES` shows the values of some MySQL system variables. This information also can be obtained using the `mysqladmin variables` command.

The `GLOBAL` and `SESSION` options are new in MySQL 4.0.3. With `GLOBAL`, you will get the values that will be used for new connections to MySQL. With `SESSION`, you will get the values that are in effect for the current connection. If you use neither option, the default is `SESSION`. `LOCAL` is a synonym for `SESSION`.

If the default values are unsuitable, you can set most of these variables using command-line options when `mysqld` starts or at runtime with the `SET` statement. See Section 6.5.3.1, "`SET` Syntax."

Partial output is shown here. The list of variables and their values may be different for your server. The meaning of each variable is given in the *MySQL Administrator's Guide*, as is information about tuning them.

```
mysql> SHOW VARIABLES;
+----------------------------------+-------------------------------+
| Variable_name                    | Value                         |
+----------------------------------+-------------------------------|
| back_log                         | 50                            |
| basedir                          | /usr/local/mysql              |
| bdb_cache_size                   | 8388572                       |
| bdb_log_buffer_size              | 32768                         |
| bdb_home                         | /usr/local/mysql              |
...
| max_connections                  | 100                           |
| max_connect_errors               | 10                            |
| max_delayed_threads              | 20                            |
| max_error_count                  | 64                            |
| max_heap_table_size              | 16777216                      |
| max_join_size                    | 4294967295                    |
| max_relay_log_size               | 0                             |
| max_sort_length                  | 1024                          |
...
| timezone                         | EEST                          |
| tmp_table_size                   | 33554432                      |
| tmpdir                           | /tmp/:/mnt/hd2/tmp/           |
| version                          | 4.0.4-beta                    |
| wait_timeout                     | 28800                         |
+----------------------------------+-------------------------------+
```

With a `LIKE` clause, the statement displays only those variables that match the pattern:

```
mysql> SHOW VARIABLES LIKE 'have%';
+--------------------+----------+
| Variable_name      | Value    |
+--------------------+----------+
| have_bdb           | YES      |
| have_innodb        | YES      |
| have_isam          | YES      |
| have_raid          | NO       |
| have_symlink       | DISABLED |
| have_openssl       | YES      |
| have_query_cache   | YES      |
+--------------------+----------+
```

#### 6.5.3.20 `SHOW WARNINGS` Syntax

```
SHOW WARNINGS [LIMIT [offset,] row_count]
SHOW COUNT(*) WARNINGS
```

`SHOW WARNINGS` shows the error, warning, and note messages that resulted from the last statement that generated messages, or nothing if the last statement that used a table generated no messages. This statement is implemented as of MySQL 4.1.0. A related statement, `SHOW ERRORS`, shows only the errors. See Section 6.5.3.9, "`SHOW ERRORS` Syntax."

The list of messages is reset for each new statement that uses a table.

The `SHOW COUNT(*) WARNINGS` statement displays the total number of errors, warnings, and notes. You can also retrieve this number from the `warning_count` variable:

```
SHOW COUNT(*) WARNINGS;
SELECT @@warning_count;
```

The value of `warning_count` might be greater than the number of messages displayed by `SHOW WARNINGS` if the `max_error_count` system variable is set low enough that not all messages are stored. An example shown later in this section demonstrates how this can happen.

The `LIMIT` clause has the same syntax as for the `SELECT` statement. See Section 6.1.7, "`SELECT` Syntax."

The MySQL server sends back the total number of errors, warnings, and notes resulting from the last statement. If you are using the C API, this value can be obtained by calling `mysql_warning_count()`.

Note that the framework for warnings was added in MySQL 4.1.0, at which point many statements did not generate warnings. In 4.1.1, the situation is much improved, with warnings generated for statements such as `LOAD DATA INFILE` and DML statements such as `INSERT`, `UPDATE`, `CREATE TABLE`, and `ALTER TABLE`.

The following `DROP TABLE` statement results in a note:

```
mysql> DROP TABLE IF EXISTS no_such_table;
mysql> SHOW WARNINGS;
+-------+------+-------------------------------+
| Level | Code | Message                       |
+-------+------+-------------------------------+
| Note  | 1051 | Unknown table 'no_such_table' |
+-------+------+-------------------------------+
```

Here is a simple example that shows a syntax warning for `CREATE TABLE` and conversion warnings for `INSERT`:

```
mysql> CREATE TABLE t1 (a TINYINT NOT NULL, b CHAR(4)) TYPE=MyISAM;
Query OK, 0 rows affected, 1 warning (0.00 sec)
mysql> SHOW WARNINGS\G
```

```
*************************** 1. row ***************************
  Level: Warning
   Code: 1287
Message: 'TYPE=storage_engine' is deprecated, use
         'ENGINE=storage_engine' instead
1 row in set (0.00 sec)

mysql> INSERT INTO t1 VALUES(10,'mysql'),(NULL,'test'),
    -> (300,'open source');
Query OK, 3 rows affected, 4 warnings (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 4

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
   Code: 1265
Message: Data truncated for column 'b' at row 1
*************************** 2. row ***************************
  Level: Warning
   Code: 1263
Message: Data truncated, NULL supplied to NOT NULL column 'a' at row 2
*************************** 3. row ***************************
  Level: Warning
   Code: 1264
Message: Data truncated, out of range for column 'a' at row 3
*************************** 4. row ***************************
  Level: Warning
   Code: 1265
Message: Data truncated for column 'b' at row 3
4 rows in set (0.00 sec)
```

The maximum number of error, warning, and note messages to store is controlled by the `max_error_count` system variable. By default, its value is 64. To change the number of messages you want stored, change the value of `max_error_count`. In the following example, the `ALTER TABLE` statement produces three warning messages, but only one is stored because `max_error_count` has been set to `1`:

```
mysql> SHOW VARIABLES LIKE 'max_error_count';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_error_count | 64    |
+-----------------+-------+
1 row in set (0.00 sec)

mysql> SET max_error_count=1;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> ALTER TABLE t1 MODIFY b CHAR;
Query OK, 3 rows affected, 3 warnings (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 3

mysql> SELECT @@warning_count;
+-----------------+
| @@warning_count |
+-----------------+
|               3 |
+-----------------+
1 row in set (0.01 sec)

mysql> SHOW WARNINGS;
+---------+------+---------------------------------------+
| Level   | Code | Message                               |
+---------+------+---------------------------------------+
| Warning | 1263 | Data truncated for column 'b' at row 1 |
+---------+------+---------------------------------------+
1 row in set (0.00 sec)
```

To disable warnings, set `max_error_count` to `0`. In this case, `warning_count` still indicates how many warnings have occurred, but none of the messages are stored.

## 6.5.4 Other Administrative Statements

### 6.5.4.1 CACHE INDEX Syntax

```
CACHE INDEX
  tbl_index_list [, tbl_index_list] ...
  IN key_cache_name

tbl_index_list:
  tbl_name [[INDEX] (index_name[, index_name] ...)]
```

The `CACHE INDEX` statement assigns table indexes to a specific key cache. It is used only for `MyISAM` tables.

The following statement assigns indexes from the tables `t1`, `t2`, and `t3` to the key cache named `hot_cache`:

```
mysql> CACHE INDEX t1, t2, t3 IN hot_cache;
+---------+--------------------+----------+----------+
| Table   | Op                 | Msg_type | Msg_text |
+---------+--------------------+----------+----------+
| test.t1 | assign_to_keycache | status   | OK       |
| test.t2 | assign_to_keycache | status   | OK       |
| test.t3 | assign_to_keycache | status   | OK       |
+---------+--------------------+----------+----------+
```

The syntax of `CACHE INDEX` allows you to specify that only particular indexes from a table should be assigned to the cache. However, the current implementation assigns all the table's indexes to the cache, so there is no reason to specify anything other than the table name.

The key cache referred to in a `CACHE INDEX` statement can be created by setting its size with a parameter setting statement or in the server parameter settings. For example:

```
mysql> SET GLOBAL keycache1.key_buffer_size=128*1024;
```

Key cache parameters can be accessed as members of a structured system variable. See Section 2.4.1, "Structured System Variables."

A key cache must exist before you can assign indexes to it:

```
mysql> CACHE INDEX t1 in non_existent_cache;
ERROR 1283 (HY000): Unknown key cache 'non_existent_cache'
```

By default, table indexes are assigned to the main (default) key cache created at the server startup. When a key cache is destroyed, all indexes assigned to it become assigned to the default key cache again.

Index assignment affects the server globally: If one client assigns an index to a given cache, this cache is used for all queries involving the index, no matter what client issues the queries.

`CACHE INDEX` was added in MySQL 4.1.1.

### 6.5.4.2 FLUSH Syntax

```
FLUSH [LOCAL | NO_WRITE_TO_BINLOG] flush_option [, flush_option] ...
```

You should use the `FLUSH` statement 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 `Host ... is blocked`. When more than `max_connect_errors` errors occur successively for a given host while connecting to the MySQL server, MySQL assumes that something is wrong and blocks the host from further connection requests. Flushing the host tables allows the host to attempt to connect again. You can start `mysqld` with `--max_connect_errors=999999999` to avoid this error message.

- `DES_KEY_FILE`

  Reloads the DES keys from the file that was specified with the `--des-key-file` option at server startup time.

- `LOGS`

  Closes and reopens all log files. If you have specified an update log file or a binary log file without an extension, the extension number of the log file will be incremented by one relative to the previous file. If you have used an extension in the file name, MySQL will close and reopen the update log or binary log file. On Unix, this is the same thing as sending a `SIGHUP` signal to the `mysqld` server.

- `PRIVILEGES`

  Reloads the privileges from the grant tables in the `mysql` database.

- `QUERY CACHE`

  Defragment the query cache to better utilize its memory. This statement does not remove any queries from the cache, unlike `RESET QUERY CACHE`.

- `STATUS`

  Resets most status variables to zero. This is something you should use only when debugging a query. See Section 1.7.1.3, "How to Report Bugs or Problems."

- `{TABLE | TABLES} [`*`tbl_name`*` [, `*`tbl_name`*`] ...]`

  When no tables are named, closes all open tables and forces all tables in use to be closed. This also flushes the query cache. With one or more table names, flushes only the given tables. `FLUSH TABLES` also removes all query results from the query cache, like the `RESET QUERY CACHE` statement.

- `TABLES WITH READ LOCK`

  Closes all open tables and locks all tables for all databases with a read lock until you execute `UNLOCK TABLES`. This is a very convenient way to get backups if you have a filesystem such as Veritas that can take snapshots in time.

- `USER_RESOURCES`

  Resets all user resources to zero. This enables clients that have reached their hourly connection, query, or update limits to resume activity. See Section 6.5.1.2, "`GRANT` and `REVOKE` Syntax."

Before MySQL 4.1.1, `FLUSH` statements are not written to the binary log. As of MySQL 4.1.1, they are written to the binary log unless the optional `NO_WRITE_TO_BINLOG` keyword (or its alias `LOCAL`) is used. Exceptions are that `FLUSH LOGS`, `FLUSH MASTER`, `FLUSH SLAVE`, and `FLUSH TABLES WITH READ LOCK` are not logged in any case because they would cause problems if replicated to a slave.

You can also access some of these statements with the `mysqladmin` utility, using the `flush-hosts`, `flush-logs`, `flush-privileges`, `flush-status`, or `flush-tables` commands.

Take also a look at the `RESET` statement used with replication. See Section 6.5.4.5, "`RESET` Syntax."

### 6.5.4.3 KILL Syntax

```
KILL [CONNECTION | QUERY] thread_id
```

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

As of MySQL 5.0.0, `KILL` allows the optional `CONNECTION` or `QUERY` modifiers:

- `KILL CONNECTION` is the same as `KILL` with no modifier: It terminates the connection associated with the given *thread_id*.
- `KILL QUERY` terminates the statement that the connection currently is executing, but leaves the connection intact.

If you have the `PROCESS` privilege, you can see all threads. If you have the `SUPER` privilege, you can kill all threads and statements. Otherwise, you can see and kill only your own threads and statements.

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

**Note:** You currently cannot use `KILL` with the Embedded MySQL Server library, because the embedded server merely runs inside the threads of the host application, it does not create connection threads of its own.

When you do a `KILL`, a thread-specific kill flag is set for the thread. In most cases, it might take some time for the thread to die, because the kill flag is checked only at specific intervals:

- In `SELECT`, `ORDER BY` and `GROUP BY` loops, the flag is checked after reading a block of rows. If the kill flag is set, the statement is aborted.
- During `ALTER TABLE`, the kill flag is checked before each block of rows is read from the original table. If the kill flag was set, the statement is aborted and the temporary table is deleted.
- During `UPDATE` or `DELETE`, the kill flag is checked after each block read and after each updated or deleted row. If the kill flag is set, the statement is aborted. Note that if you are not using transactions, the changes will not be rolled back!
- `GET_LOCK()` will abort and return `NULL`.
- An `INSERT DELAYED` thread will quickly flush all rows it has in memory and terminate.
- If the thread is in the table lock handler (state: `Locked`), the table lock will be quickly aborted.
- If the thread is waiting for free disk space in a write call, the write is aborted with a "disk full" error message.

### 6.5.4.4 LOAD INDEX INTO CACHE Syntax

```
LOAD INDEX INTO CACHE
  tbl_index_list [, tbl_index_list] ...

tbl_index_list:
  tbl_name
    [[INDEX] (index_name[, index_name] ...)]
    [IGNORE LEAVES]
```

The `LOAD INDEX INTO CACHE` statement preloads a table index into the key cache to which it has been assigned by an explicit `CACHE INDEX` statement, or into the default key cache otherwise. `LOAD INDEX INTO CACHE` is used only for `MyISAM` tables.

The `IGNORE LEAVES` modifier causes only blocks for the non-leaf nodes of the index to be preloaded.

The following statement preloads nodes (index blocks) of indexes of the tables `t1` and `t2`:

```
mysql> LOAD INDEX INTO CACHE t1, t2 IGNORE LEAVES;
+---------+--------------+----------+----------+
| Table   | Op           | Msg_type | Msg_text |
+---------+--------------+----------+----------+
| test.t1 | preload_keys | status   | OK       |
| test.t2 | preload_keys | status   | OK       |
+---------+--------------+----------+----------+
```

This statement preloads all index blocks from `t1`. It preloads only blocks for the non-leaf nodes from `t2`.

The syntax of `LOAD INDEX INTO CACHE` allows you to specify that only particular indexes from a table should be preloaded. However, the current implementation preloads all the table's indexes into the cache, so there is no reason to specify anything other than the table name.

`LOAD INDEX INTO CACHE` was added in MySQL 4.1.1.

### 6.5.4.5 RESET Syntax

```
RESET reset_option [, reset_option] ...
```

The `RESET` statement is used to clear the state of various server operations. It also acts as a stronger version of the `FLUSH` statement. See Section 6.5.4.2, "`FLUSH` Syntax."

To execute `RESET`, you must have the `RELOAD` privilege.

*`reset_option`* can be any of the following:

- `MASTER`

  Deletes all binary logs listed in the index file, resets the binary log index file to be empty, and creates a new binary log file. Previously named `FLUSH MASTER`. See Section 6.6.1, "SQL Statements for Controlling Master Servers."

- `QUERY CACHE`

  Removes all query results from the query cache.
- `SLAVE`

  Makes the slave forget its replication position in the master binary logs. Previously named `FLUSH SLAVE`. See Section 6.6.2, "SQL Statements for Controlling Slave Servers."

## 6.6 Replication Statements

This section describes replication-related SQL statements. One group of statements is used for controlling master servers. The other is used for controlling slave servers.

### 6.6.1 SQL Statements for Controlling Master Servers

Replication can be controlled through the SQL interface. This section discusses statements for managing master replication servers. Section 6.6.2, "SQL Statements for Controlling Slave Servers," discusses statements for managing slave servers.

#### 6.6.1.1 `PURGE MASTER LOGS` Syntax

```
PURGE {MASTER | BINARY} LOGS TO 'log_name'
PURGE {MASTER | BINARY} LOGS BEFORE 'date'
```

Deletes all the binary logs listed in the log index that are strictly prior to the specified log or date. The logs also are removed from the list recorded in the log index file, so that the given log becomes the first.

Examples:

```
PURGE MASTER LOGS TO 'mysql-bin.010';
PURGE MASTER LOGS BEFORE '2003-04-02 22:46:26';
```

The `BEFORE` variant is available as of MySQL 4.1. Its date argument can be in `'YYYY-MM-DD hh:mm:ss'` format. `MASTER` and `BINARY` are synonyms, but `BINARY` can be used only as of MySQL 4.1.1.

If you have an active slave that currently is reading one of the logs you are trying to delete, this statement does nothing and fails with an error. However, if a slave is dormant and you happen to purge one of the logs it wants to read, the slave will be unable to replicate once it comes up. The statement is safe to run while slaves are replicating. You do not need to stop them.

To purge logs, follow this procedure:

1. On each slave server, use `SHOW SLAVE STATUS` to check which log it is reading.
2. Obtain a listing of the logs on the master server with `SHOW MASTER LOGS`.
3. Determine the earliest log among all the slaves. This is the target log. If all the slaves are up to date, this will be the last log on the list.
4. Make a backup of all the logs you are about to delete. (The step is optional, but a good idea.)
5. Purge all logs up to but not including the target log.

#### 6.6.1.2 `RESET MASTER` Syntax

```
RESET MASTER
```

Deletes all binary logs listed in the index file, resets the binary log index file to be empty, and creates a new binary log file.

This statement was named `FLUSH MASTER` before MySQL 3.23.26.

#### 6.6.1.3 `SET SQL_LOG_BIN` Syntax

```
SET SQL_LOG_BIN = {0|1}
```

Disables or enables binary logging for the current connection (`SQL_LOG_BIN` is a session variable) if the client connects using an account that has the `SUPER` privilege. The statement is refused with an error if the client does not have that privilege. (Before MySQL 4.1.2, the statement was simply ignored in that case.)

#### 6.6.1.4 `SHOW BINLOG EVENTS` Syntax

```
SHOW BINLOG EVENTS
   [IN 'log_name'] [FROM pos] [LIMIT [offset,] row_count]
```

Shows the events in the binary log. If you do not specify `'log_name'`, the first binary log is displayed.

The `LIMIT` clause has the same syntax as for the `SELECT` statement. See Section 6.1.7, "`SELECT` Syntax."

This statement is available as of MySQL 4.0.

#### 6.6.1.5 `SHOW MASTER LOGS` Syntax

```
SHOW MASTER LOGS
```

Lists the binary log files on the master. This statement is used as part of the procedure described in Section 6.6.1.1, "`PURGE MASTER LOGS` Syntax," for determining which logs can be purged.

### 6.6.1.6 `SHOW MASTER STATUS` Syntax

```
SHOW MASTER STATUS
```

Provides status information on the binary log files of the master.

### 6.6.1.7 `SHOW SLAVE HOSTS` Syntax

```
SHOW SLAVE HOSTS
```

Displays a list of slaves currently registered with the master. Any slave not started with the `--report-host=slave_name` option will not be visible in that list.

## 6.6.2 SQL Statements for Controlling Slave Servers

Replication can be controlled through the SQL interface. This section discusses statements for managing slave replication servers. Section 6.6.1, "SQL Statements for Controlling Master Servers," discusses statements for managing master servers.

### 6.6.2.1 `CHANGE MASTER TO` Syntax

```
CHANGE MASTER TO master_def [, master_def] ...

master_def:
      MASTER_HOST = 'host_name'
    | MASTER_USER = 'user_name'
    | MASTER_PASSWORD = 'password'
    | MASTER_PORT = port_num
    | MASTER_CONNECT_RETRY = count
    | MASTER_LOG_FILE = 'master_log_name'
    | MASTER_LOG_POS = master_log_pos
    | RELAY_LOG_FILE = 'relay_log_name'
    | RELAY_LOG_POS = relay_log_pos
    | MASTER_SSL = {0|1}
    | MASTER_SSL_CA = 'ca_file_name'
    | MASTER_SSL_CAPATH = 'ca_directory_name'
    | MASTER_SSL_CERT = 'cert_file_name'
    | MASTER_SSL_KEY = 'key_file_name'
    | MASTER_SSL_CIPHER = 'cipher_list'
```

Changes the parameters that the slave server uses for connecting to and communicating with the master server.

`MASTER_USER`, `MASTER_PASSWORD`, `MASTER_SSL`, `MASTER_SSL_CA`, `MASTER_SSL_CAPATH`, `MASTER_SSL_CERT`, `MASTER_SSL_KEY`, and `MASTER_SSL_CIPHER` provide information for the slave about how to connect to its master.

The relay log options (`RELAY_LOG_FILE` and `RELAY_LOG_POS`) are available beginning with MySQL 4.0.

The SSL options (MASTER_SSL, MASTER_SSL_CA, MASTER_SSL_CAPATH, MASTER_SSL_CERT, MASTER_SSL_KEY, and MASTER_SSL_CIPHER) are available beginning with MySQL 4.1.1. You can change these options even on slaves that are compiled without SSL support. They are saved to the master.info file, but are ignored until you use a server that has SSL support enabled.

If you don't specify a given parameter, it keeps its old value, except as indicated in the following discussion. For example, if the password to connect to your MySQL master has changed, you just need to issue these statements to tell the slave about the new password:

```
mysql> STOP SLAVE; -- if replication was running
mysql> CHANGE MASTER TO MASTER_PASSWORD='new3cret';
mysql> START SLAVE; -- if you want to restart replication
```

There is no need to specify the parameters that do not change (host, port, user, and so forth).

MASTER_HOST and MASTER_PORT are the hostname (or IP address) of the master host and its TCP/IP port. Note that if MASTER_HOST is equal to localhost, then, like in other parts of MySQL, the port may be ignored (if Unix socket files can be used, for example).

If you specify MASTER_HOST or MASTER_PORT, the slave assumes that the master server is different than before (even if you specify a host or port value that is the same as the current value). In this case, the old values for the master binary log name and position are considered no longer applicable, so if you do not specify MASTER_LOG_FILE and MASTER_LOG_POS in the statement, MASTER_LOG_FILE='' and MASTER_LOG_POS=4 are silently appended to it.

MASTER_LOG_FILE and MASTER_LOG_POS are the coordinates at which the slave I/O thread should begin reading from the master the next time the thread starts. If you specify either of them, you can't specify RELAY_LOG_FILE or RELAY_LOG_POS. If neither of MASTER_LOG_FILE or MASTER_LOG_POS are specified, the slave uses last coordinates of the *slave SQL thread* before CHANGE MASTER was issued. This ensures that replication has no discontinuity, even if the slave SQL thread was late compared to the slave I/O thread, when you just want to change, say, the password to use. This safe behavior was introduced starting from MySQL 4.0.17 and 4.1.1. (Before these versions, the coordinates used were the last coordinates of the slave I/O thread before CHANGE MASTER was issued. This caused the SQL thread to possibly lose some events from the master, thus breaking replication.)

CHANGE MASTER *deletes all relay log files* and starts a new one, unless you specify RELAY_LOG_FILE or RELAY_LOG_POS. In that case, relay logs are kept; as of MySQL 4.1.1, the relay_log_purge global variable is silently set to 0.

CHANGE MASTER TO updates the contents of the master.info and relay-log.info files.

CHANGE MASTER is useful for setting up a slave when you have the snapshot of the master and have recorded the log and the offset corresponding to it. After loading the snapshot into the slave, you can run CHANGE MASTER TO MASTER_LOG_FILE='*log_name_on_master*', MASTER_LOG_POS=*log_offset_on_master* on the slave.

Examples:

```
mysql> CHANGE MASTER TO
    ->     MASTER_HOST='master2.mycompany.com',
    ->     MASTER_USER='replication',
    ->     MASTER_PASSWORD='bigs3cret',
    ->     MASTER_PORT=3306,
    ->     MASTER_LOG_FILE='master2-bin.001',
    ->     MASTER_LOG_POS=4,
    ->     MASTER_CONNECT_RETRY=10;

mysql> CHANGE MASTER TO
    ->     RELAY_LOG_FILE='slave-relay-bin.006',
    ->     RELAY_LOG_POS=4025;
```

The first example changes the master and master's binary log coordinates. This is used when you want to set up the slave to replicate the master.

The second example shows an operation that is less frequently used. It is done when the slave has relay logs that you want it to execute again for some reason. To do this, the master need not be reachable. You just have to use `CHANGE MASTER TO` and start the SQL thread (`START SLAVE SQL_THREAD`).

You can even use the second operation in a non-replication setup with a standalone, non-slave server, to recover after a crash. Suppose that your server has crashed and you have restored a backup. You want to replay the server's own binary logs (not relay logs, but regular binary logs), supposedly named `myhost-bin.*`. First, make a backup copy of these binary logs in some safe place, in case you don't exactly follow the procedure below and accidentally have the server purge the binary logs. If using MySQL 4.1.1 or newer, use `SET GLOBAL relay_log_purge=0` for additional safety. Then start the server without the `--log-bin` option, with a new (different from before) server ID, with `--relay-log=myhost-bin` (to make the server believe that these regular binary logs are relay logs) and with `--skip-slave-start`. After the server starts, issue these statements:

```
mysql> CHANGE MASTER TO
    ->     RELAY_LOG_FILE='myhost-bin.153',
    ->     RELAY_LOG_POS=410,
    ->     MASTER_HOST='some_dummy_string';
mysql> START SLAVE SQL_THREAD;
```

The server will read and execute its own binary logs, thus achieving crash recovery. Once the recovery is finished, run `STOP SLAVE`, shut down the server, delete `master.info` and `relay-log.info`, and restart the server with its original options.

For the moment, specifying `MASTER_HOST` (even with a dummy value) is required to make the server think it is a slave. Giving the server a new, different from before, server ID is also required or the server will see events with its ID and think it is in a circular replication setup and skip the events, which is unwanted. In the future, we plan to add options to get rid of these small constraints.

### 6.6.2.2 LOAD DATA FROM MASTER Syntax

```
LOAD DATA FROM MASTER
```

Takes a snapshot of the master and copies it to the slave. It updates the values of MASTER_LOG_FILE and MASTER_LOG_POS so that the slave will start replicating from the correct position. Any table and database exclusion rules specified with the --replicate-*-do-* and --replicate-*-ignore-* options are honored. --replicate-rewrite-db is not taken into account (because one user could, with this option, set up a non-unique mapping *such as* --replicate-rewrite-db=db1->db3 and --replicate-rewrite-db=db2->db3, which would confuse the slave when it loads the master's tables).

Use of this statement is subject to the following conditions:

- It works only with MyISAM tables.
- It acquires a global read lock on the master while taking the snapshot, which prevents updates on the master during the load operation.

In the future, it is planned to make this statement work with InnoDB tables and to remove the need for a global read lock by using non-blocking online backup.

If you are loading big tables, you might have to increase the values of net_read_timeout and net_write_timeout on both your master and slave servers.

Note that LOAD DATA FROM MASTER does *not* copy any tables from the mysql database. This makes it easy to have different users and privileges on the master and the slave.

The LOAD DATA FROM MASTER statement requires the replication account that is used to connect to the master to have the RELOAD and SUPER privileges on the master and the SELECT privilege for all master tables you want to load. All master tables for which the user does not have the SELECT privilege are ignored by LOAD DATA FROM MASTER. This is because the master will hide them from the user: LOAD DATA FROM MASTER calls SHOW DATABASES to know the master databases to load, but SHOW DATABASES returns only databases for which the user has some privilege. See Section 6.5.3.7, "SHOW DATABASES Syntax." On the slave's side, the user that issues LOAD DATA FROM MASTER should have grants to drop and create the databases and tables that are copied.

### 6.6.2.3 LOAD TABLE *tbl_name* FROM MASTER Syntax

```
LOAD TABLE tbl_name FROM MASTER
```

Transfers a copy of the table from master to the slave. This statement is implemented mainly for debugging of LOAD DATA FROM MASTER. It requires that the account used for connecting to the master server has the RELOAD and SUPER privileges on the master and the SELECT privilege on the master table to load. On the slave side, the user that issues LOAD TABLE FROM MASTER should have privileges to drop and create the table.

The conditions for LOAD DATA FROM MASTER apply here, too. For example, LOAD TABLE FROM MASTER works only for MyISAM tables. The timeout notes for LOAD DATA FROM MASTER apply as well.

### 6.6.2.4 `MASTER_POS_WAIT()` Syntax

```
SELECT MASTER_POS_WAIT('master_log_file', master_log_pos)
```

This is a function, not a statement. It is used to ensure that the slave has read and executed events up to a given position in the master's binary log. See Section 5.8.4, "Miscellaneous Functions," for a full description.

### 6.6.2.5 `RESET SLAVE` Syntax

```
RESET SLAVE
```

Makes the slave forget its replication position in the master's binary logs. This statement is meant to be used for a clean start: It deletes the `master.info` and `relay-log.info` files, all the relay logs, and starts a new relay log.

**Note:** All relay logs are deleted, even if they have not been totally executed by the slave SQL thread. (This is a condition likely to exist on a replication slave if you have issued a `STOP SLAVE` statement or if the slave is highly loaded.)

Connection information stored in the `master.info` file is immediately reset using any values specified in the corresponding startup options. This information includes values such as master host, master port, master user, and master password. If the slave SQL thread was in the middle of replicating temporary tables when it was stopped, and `RESET SLAVE` is issued, these replicated temporary tables are deleted on the slave.

This statement was named `FLUSH SLAVE` before MySQL 3.23.26.

### 6.6.2.6 `SET GLOBAL SQL_SLAVE_SKIP_COUNTER` Syntax

```
SET GLOBAL SQL_SLAVE_SKIP_COUNTER = n
```

Skip the next *n* events from the master. This is useful for recovering from replication stops caused by a statement.

This statement is valid only when the slave thread is not running. Otherwise, it produces an error.

Before MySQL 4.0, omit the `GLOBAL` keyword from the statement.

### 6.6.2.7 `SHOW SLAVE STATUS` Syntax

```
SHOW SLAVE STATUS
```

Provides status information on essential parameters of the slave threads. If you issue this statement using the `mysql` client, you can use a `\G` statement terminator rather than semicolon to get a more readable vertical layout:

```
mysql> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
       Slave_IO_State: Waiting for master to send event
          Master_Host: localhost
          Master_User: root
          Master_Port: 3306
        Connect_Retry: 3
      Master_Log_File: gbichot-bin.005
  Read_Master_Log_Pos: 79
       Relay_Log_File: gbichot-relay-bin.005
        Relay_Log_Pos: 548
Relay_Master_Log_File: gbichot-bin.005
     Slave_IO_Running: Yes
    Slave_SQL_Running: Yes
      Replicate_Do_DB:
  Replicate_Ignore_DB:
           Last_Errno: 0
           Last_Error:
         Skip_Counter: 0
  Exec_Master_Log_Pos: 79
      Relay_Log_Space: 552
      Until_Condition: None
       Until_Log_File:
        Until_Log_Pos: 0
   Master_SSL_Allowed: No
   Master_SSL_CA_File:
   Master_SSL_CA_Path:
      Master_SSL_Cert:
    Master_SSL_Cipher:
       Master_SSL_Key:
Seconds_Behind_Master: 8
```

Depending on your version of MySQL, you may not see all the fields just shown. In particular, several fields are present only as of MySQL 4.1.1.

`SHOW SLAVE STATUS` returns the following fields:

- `Slave_IO_State`

  A copy of the `State` field of the output of `SHOW PROCESSLIST` for the slave I/O thread. This tells you if the thread is trying to connect to the master, waiting for events from the master, reconnecting to the master, and so on. Looking at this field is necessary because, for example, the thread can be running but unsuccessfully trying to connect to the master; only this field will make you aware of the connection problem. The state of the SQL thread is not copied because it is simpler. If it is running, there is no problem; if it is not, you will find the error in the `Last_Error` field (described below).

  This field is present beginning with MySQL 4.1.1.

- `Master_Host`

  The current master host.

- `Master_User`

  The current user used to connect to the master.

- `Master_Port`

  The current master port.

- `Connect_Retry`

  The current value of the `--master-connect-retry` option.

- `Master_Log_File`

  The name of the master binary log file from which the I/O thread is currently reading.

- `Read_Master_Log_Pos`

  The position up to which the I/O thread has read in the current master binary log.

- `Relay_Log_File`

  The name of the relay log file from which the SQL thread is currently reading and executing.

- `Relay_Log_Pos`

  The position up to which the SQL thread has read and executed in the current relay log.

- `Relay_Master_Log_File`

  The name of the master binary log file that contains the last event executed by the SQL thread.

- `Slave_IO_Running`

  Whether or not the I/O thread is started.

- `Slave_SQL_Running`

  Whether or not the SQL thread is started.

- `Replicate_Do_DB, Replicate_Ignore_DB`

  The lists of databases that were specified with the `--replicate-do-db` and `--replicate-ignore-db` options, if any.

  These fields are present beginning with MySQL 4.1.1.

- `Replicate_Do_Table, Replicate_Ignore_Table, Replicate_Wild_Do_Table, Replicate_Wild_Ignore_Table`

  The lists of tables that were specified with the `--replicate-do-table`, `--replicate-ignore-table`, `--replicate-wild-do-table`, and `--replicate-wild-ignore_table` options, if any.

  These fields are present beginning with MySQL 4.1.1.

- `Last_Errno, Last_Error`

  The error number and error message returned by the most recently executed query. An error number of 0 and message of the empty string mean "no error." If the `Last_Error` value is not empty, it will also appear as a message in the slave's error log.

  For example:

  ```
  Last_Errno: 1051
  Last_Error: error 'Unknown table 'z'' on query 'drop table z'
  ```

  The message indicates that the table `z` existed on the master and was dropped there, but it did not exist on the slave, so `DROP TABLE` failed on the slave. (This might occur, for example, if you forget to copy the table to the slave when setting up replication.)

- `Skip_Counter`

  The last used value for `SQL_SLAVE_SKIP_COUNTER`.

- `Exec_Master_Log_Pos`

  The position of the last event executed by the SQL thread from the master's binary log (`Relay_Master_Log_File`). (`Relay_Master_Log_File`, `Exec_Master_Log_Pos`) in the master's binary log corresponds to (`Relay_Log_File`, `Relay_Log_Pos`) in the relay log.

- `Relay_Log_Space`

  The total combined size of all existing relay logs.

- `Until_Condition, Until_Log_File, Until_Log_Pos`

  The values specified in the `UNTIL` clause of the `START SLAVE` statement.

  `Until_Condition` has these values:

  - `None` if no `UNTIL` clause was specified
  - `Master` if the slave is reading until a given position in the master's binary logs
  - `Relay` if the slave is reading until a given position in its relay logs

  `Until_Log_File` and `Until_Log_Pos` indicate the log filename and position values that define the point at which the SQL thread will stop executing.

  These fields are present beginning with MySQL 4.1.1.

- `Master_SSL_Allowed`, `Master_SSL_CA_File`, `Master_SSL_CA_Path`, `Master_SSL_Cert`, `Master_SSL_Cipher`, `Master_SSL_Key`

  These fields show the SSL parameters used by the slave to connect to the master, if any.

  `Master_SSL_Allowed` has these values:

  - `Yes` if an SSL connection to the master is allowed
  - `No` if an SSL connection to the master is not allowed
  - `Ignored` if an SSL connection is allowed but the slave server does not have SSL support enabled

  The values of the other SSL-related fields correspond to the values of the `--master-ca`, `--master-capath`, `--master-cert`, `--master-cipher`, and `--master-key` options.

  These fields are present beginning with MySQL 4.1.1.

- `Seconds_Behind_Master`

  The number of seconds that have elapsed since the timestamp of the last master's event executed by the slave SQL thread. This will be `NULL` when no event has been executed yet, or after `CHANGE MASTER` and `RESET SLAVE`. This field can be used to know how "late" your slave is. It will work even though your master and slave don't have identical clocks.

  This field is present beginning with MySQL 4.1.1.

### 6.6.2.8 START SLAVE Syntax

```
START SLAVE [thread_type [, thread_type] ... ]
START SLAVE [SQL_THREAD] UNTIL
    MASTER_LOG_FILE = 'log_name', MASTER_LOG_POS = log_pos
START SLAVE [SQL_THREAD] UNTIL
    RELAY_LOG_FILE = 'log_name', RELAY_LOG_POS = log_pos

thread_type: IO_THREAD | SQL_THREAD
```

`START SLAVE` with no options starts both of the slave threads. The I/O thread reads queries from the master server and stores them in the relay log. The SQL thread reads the relay log and executes the queries. `START SLAVE` requires the `SUPER` privilege.

If `START SLAVE` succeeds in starting the slave threads, it returns without any error. However, even in that case, it might be that the slave threads start and then later stop (for example, because they don't manage to connect to the master or read its binary logs, or some other problem). `START SLAVE` will not warn you about this. You must check your slave's error log for error messages generated by the slave threads, or check that they are running fine with `SHOW SLAVE STATUS`.

As of MySQL 4.0.2, you can add `IO_THREAD` and `SQL_THREAD` options to the statement to name which of the threads to start.

As of MySQL 4.1.1, an UNTIL clause may be added to specify that the slave should start and run until the SQL thread reaches a given point in the master binary logs or in the slave relay logs. When the SQL thread reaches that point, it stops. If the SQL_THREAD option is specified in the statement, it starts only the SQL thread. Otherwise, it starts both slave threads. If the SQL thread is already running, the UNTIL clause is ignored and a warning is issued.

With an UNTIL clause, you must specify both a log filename and position. Do not mix master and relay log options.

Any UNTIL condition is reset by a subsequent STOP SLAVE statement, a START SLAVE statement that includes no UNTIL clause, or a server restart.

The UNTIL clause can be useful for debugging replication, or to cause replication to proceed until just before the point where you want to avoid having the slave replicate a statement. For example, if an unwise DROP TABLE statement was executed on the master, you can use UNTIL to tell the slave to execute up to that point but no farther. To find what the event is, use mysqlbinlog with the master logs or slave relay logs, or by using a SHOW BINLOG EVENTS statement.

If you are using UNTIL to have the slave process replicated queries in sections, it is recommended that you start the slave with the --skip-slave-start option to prevent the SQL thread from running when the slave server starts. It is probably best to use this option in an option file rather than on the command line, so that an unexpected server restart does not cause it to be forgotten.

The SHOW SLAVE STATUS statement includes output fields that display the current values of the UNTIL condition.

This statement is called SLAVE START before MySQL 4.0.5. For the moment, SLAVE START is still accepted for backward compatibility, but is deprecated.

#### 6.6.2.9 STOP SLAVE Syntax

```
STOP SLAVE [thread_type [, thread_type] ... ]

thread_type: IO_THREAD | SQL_THREAD
```

Stops the slave threads. STOP SLAVE requires the SUPER privilege.

Like START SLAVE, as of MySQL 4.0.2, this statement may be used with the IO_THREAD and SQL_THREAD options to name the thread or threads to stop.

This statement is called SLAVE STOP before MySQL 4.0.5. For the moment, SLAVE STOP is still accepted for backward compatibility, but is deprecated.

# 7 Spatial Extensions in MySQL

MySQL 4.1 introduces spatial extensions to allow the generation, storage, and analysis of geographic features. Currently, these features are available for `MyISAM` tables only.

This chapter covers the following topics:

- The basis of these spatial extensions in the OpenGIS geometry model
- Data formats for representing spatial data
- How to use spatial data in MySQL
- Use of indexing for spatial data
- MySQL differences from the OpenGIS specification

## 7.1 Introduction

MySQL implements spatial extensions following the specification of the Open GIS Consortium, Inc. (OGC). This is an international consortium of more than 250 companies, agencies, and universities participating in the development of publicly available conceptual solutions that can be useful with all kinds of applications that manage spatial data. The OGC maintains a Web site at `http://www.opengis.org/`.

In 1997, the Open GIS Consortium published the "OpenGIS ® Simple Features Specifications For SQL," a document that proposes several conceptual ways for extending an SQL RDBMS to support spatial data. This specification is available from the Open GIS Web site at `http://www.opengis.org/docs/99-049.pdf`. It contains additional information relevant to this chapter.

MySQL implements a subset of the **SQL with Geometry Types** environment proposed by OGC. This term refers to an SQL environment that has been extended with a set of geometry types. A geometry-valued SQL column is implemented as a column that has a geometry type. The specifications describe a set of SQL geometry types, as well as functions on those types to create and analyze geometry values.

A **geographic feature** is anything in the world that has a location. A feature can be:

- An entity. For example, a mountain, a pond, a city.
- A space. For example, a postcode area, the tropics.
- A definable location. For example, a crossroad, as a particular place where two streets intersect.

You can also find documents that use the term **geospatial feature** to refer to geographic features.

**Geometry** is another word that denotes a geographic feature. Originally, the word **geometry** meant measurement of the earth. Another meaning comes from cartography, referring to the geometric features that cartographers use to map the world.

This chapter uses all of these terms synonymously: **geographic feature**, **geospatial feature**, **feature**, or **geometry**. The term most commonly used here is **geometry**.

Let's define a **geometry** as *a point or an aggregate of points representing anything in the world that has a location*.

## 7.2 The OpenGIS Geometry Model

The set of geometry types proposed by OGC's **SQL with Geometry Types** environment is based on the **OpenGIS Geometry Model**. In this model, each geometric object has the following general properties:

- It is associated with a Spatial Reference System, which describes the coordinate space in which the object is defined.
- It belongs to some geometry class.

### 7.2.1 The Geometry Class Hierarchy

The geometry classes define a hierarchy as follows:

- `Geometry` (non-instantiable)
    - `Point` (instantiable)
    - `Curve` (non-instantiable)
- `LineString` (instantiable)
    - `Line`
    - `LinearRing`
- `Surface` (non-instantiable)
    - `Polygon` (instantiable)

- GeometryCollection (instantiable)
    - MultiPoint (instantiable)
    - MultiCurve (non-instantiable)
    - MultiLineString (instantiable)
- MultiSurface (non-instantiable)
    - MultiPolygon (instantiable)

It is not possible to create objects in non-instantiable classes. It is possible to create objects in instantiable classes. All classes have properties, and instantiable classes may also have assertions (rules that define valid class instances).

Geometry is the base class. It's an abstract class. The instantiable subclasses of Geometry are restricted to zero-, one-, and two-dimensional geometric objects that exist in two-dimensional coordinate space. All instantiable geometry classes are defined so that valid instances of a geometry class are topologically closed (that is, all defined geometries include their boundary).

The base Geometry class has subclasses for Point, Curve, Surface, and GeometryCollection:

- Point represents zero-dimensional objects.
- Curve represents one-dimensional objects, and has subclass LineString, with sub-sub-classes Line and LinearRing.
- Surface is designed for two-dimensional objects and has subclass Polygon.
- GeometryCollection has specialized zero-, one-, and two-dimensional collection classes named MultiPoint, MultiLineString, and MultiPolygon for modeling geometries corresponding to collections of Points, LineStrings, and Polygons, respectively. MultiCurve and MultiSurface are introduced as abstract superclasses that generalize the collection interfaces to handle Curves and Surfaces.

Geometry, Curve, Surface, MultiCurve, and MultiSurface are defined as non-instantiable classes. They define a common set of methods for their subclasses and are included for extensibility.

Point, LineString, Polygon, GeometryCollection, MultiPoint, MultiLineString, and MultiPolygon are instantiable classes.

## 7.2.2 Class Geometry

Geometry is the root class of the hierarchy. It is a non-instantiable class but has a number of properties that are common to all geometry values created from any of the Geometry subclasses. These properties are described in the following list. (Particular subclasses have their own specific properties, described later.)

## Geometry Properties

A geometry value has the following properties:

- Its **type**. Each geometry belongs to one of the instantiable classes in the hierarchy.
- Its **SRID**, or Spatial Reference Identifier. This value identifies the geometry's associated Spatial Reference System that describes the coordinate space in which the geometry object is defined.
- Its **coordinates** in its Spatial Reference System, represented as double-precision (eight-byte) numbers. All non-empty geometries include at least one pair of (X,Y) coordinates. Empty geometries contain no coordinates.

  Coordinates are related to the SRID. For example, in different coordinate systems, the distance between two objects may differ even when objects have the same coordinates, because the distance on the **planar** coordinate system and the distance on the **geocentric** system (coordinates on the Earth's surface) are different things.
- Its **interior**, **boundary**, and **exterior**.

  Every geometry occupies some position in space. The exterior of a geometry is all space not occupied by the geometry. The interior is the space occupied by the geometry. The boundary is the interface between the geometry's interior and exterior.
- Its **MBR** (Minimum Bounding Rectangle), or Envelope. This is the bounding geometry, formed by the minimum and maximum (X,Y) coordinates:

  ```
  ((MINX MINY, MAXX MINY, MAXX MAXY, MINX MAXY, MINX MINY))
  ```
- The quality of being **simple** or **non-simple**. Geometry values of types (`LineString`, `MultiPoint`, `MultiLineString`) are either simple or non-simple. Each type determines its own assertions for being simple or non-simple.
- The quality of being **closed** or **not closed**. Geometry values of types (`LineString`, `MultiString`) are either closed or not closed. Each type determines its own assertions for being closed or not closed.
- The quality of being **empty** or **not empty**. A geometry is empty if it does not have any points. Exterior, interior, and boundary of an empty geometry are not defined (that is, they are represented by a `NULL` value). An empty geometry is defined to be always simple and has an area of 0.
- Its **dimension**. A geometry can have a dimension of –1, 0, 1, or 2:
  - –1 for an empty geometry.
  - 0 for a geometry with no length and no area.
  - 1 for a geometry with non-zero length and zero area.
  - 2 for a geometry with non-zero area.

  `Point` objects have a dimension of zero. `LineString` objects have a dimension of 1. `Polygon` objects have a dimension of 2. The dimensions of `MultiPoint`, `MultiLineString`, and `MultiPolygon` objects are the same as the dimensions of the elements they consist of.

## 7.2.3 Class `Point`

A `Point` is a geometry that represents a single location in coordinate space.

### `Point` Examples

- Imagine a large-scale map of the world with many cities. A `Point` object could represent each city.
- On a city map, a `Point` object could represent a bus stop.

### `Point` Properties

- X-coordinate value.
- Y-coordinate value.
- `Point` is defined as a zero-dimensional geometry.
- The boundary of a `Point` is the empty set.

## 7.2.4 Class `Curve`

A `Curve` is a one-dimensional geometry, usually represented by a sequence of points. Particular subclasses of `Curve` define the type of interpolation between points. `Curve` is a non-instantiable class.

### `Curve` Properties

- A `Curve` has the coordinates of its points.
- A `Curve` is defined as a one-dimensional geometry.
- A `Curve` is simple if it does not pass through the same point twice.
- A `Curve` is closed if its start point is equal to its end point.
- The boundary of a closed `Curve` is empty.
- The boundary of a non-closed `Curve` consists of its two end points.
- A `Curve` that is simple and closed is a `LinearRing`.

## 7.2.5 Class `LineString`

A `LineString` is a `Curve` with linear interpolation between points.

### `LineString` Examples

- On a world map, `LineString` objects could represent rivers.
- In a city map, `LineString` objects could represent streets.

### `LineString` Properties

- A `LineString` has coordinates of segments, defined by each consecutive pair of points.
- A `LineString` is a `Line` if it consists of exactly two points.
- A `LineString` is a `LinearRing` if it is both closed and simple.

## 7.2.6 Class `Surface`

A `Surface` is a two-dimensional geometry. It is a non-instantiable class. Its only instantiable subclass is `Polygon`.

### `Surface` Properties

- A `Surface` is defined as a two-dimensional geometry.
- The OpenGIS specification defines a simple `Surface` as a geometry that consists of a single "patch" that is associated with a single exterior boundary and zero or more interior boundaries.
- The boundary of a simple `Surface` is the set of closed curves corresponding to its exterior and interior boundaries.

## 7.2.7 Class `Polygon`

A `Polygon` is a planar `Surface` representing a multisided geometry. It is defined by a single exterior boundary and zero or more interior boundaries, where each interior boundary defines a hole in the `Polygon`.

### `Polygon` Examples

- On a region map, `Polygon` objects could represent forests, districts, and so on.

### `Polygon` Assertions

- The boundary of a `Polygon` consists of a set of `LinearRing` objects (that is, `LineString` objects that are both simple and closed) that make up its exterior and interior boundaries.
- A `Polygon` has no rings that cross. The rings in the boundary of a `Polygon` may intersect at a `Point`, but only as a tangent.
- A `Polygon` has no lines, spikes, or punctures.
- A `Polygon` has an interior that is a connected point set.
- A `Polygon` may have holes. The exterior of a `Polygon` with holes is not connected. Each hole defines a connected component of the exterior.

The preceding assertions make a `Polygon` a simple geometry.

## 7.2.8 Class `GeometryCollection`

A `GeometryCollection` is a geometry that is a collection of one or more geometries of any class.

All the elements in a `GeometryCollection` must be in the same Spatial Reference System (that is, in the same coordinate system). There are no other constraints on the elements of a `GeometryCollection`, although the subclasses of `GeometryCollection` described in the following sections may restrict membership. Restrictions may be based on:

- Element type (for example, a `MultiPoint` may contain only `Point` elements)
- Dimension
- Constraints on the degree of spatial overlap between elements

## 7.2.9 Class `MultiPoint`

A `MultiPoint` is a geometry collection composed of `Point` elements. The points are not connected or ordered in any way.

### `MultiPoint` Examples

- On a world map, a `MultiPoint` could represent a chain of small islands.
- On a city map, a `MultiPoint` could represent the outlets for a ticket office.

### `MultiPoint` Properties

- A `MultiPoint` is a zero-dimensional geometry.
- A `MultiPoint` is simple if no two of its `Point` values are equal (have identical coordinate values).
- The boundary of a `MultiPoint` is the empty set.

## 7.2.10 Class `MultiCurve`

A `MultiCurve` is a geometry collection composed of `Curve` elements. `MultiCurve` is a non-instantiable class.

### `MultiCurve` Properties

- A `MultiCurve` is a one-dimensional geometry.
- A `MultiCurve` is simple if and only if all of its elements are simple; the only intersections between any two elements occur at points that are on the boundaries of both elements.

- A `MultiCurve` boundary is obtained by applying the "mod 2 union rule" (also known as the "odd-even rule"): A point is in the boundary of a `MultiCurve` if it is in the boundaries of an odd number of `MultiCurve` elements.
- A `MultiCurve` is closed if all of its elements are closed.
- The boundary of a closed `MultiCurve` is always empty.

## 7.2.11 Class `MultiLineString`

A `MultiLineString` is a `MultiCurve` geometry collection composed of `LineString` elements.

### `MultiLineString` Examples

- On a region map, a `MultiLineString` could represent a river system or a highway system.

## 7.2.12 Class `MultiSurface`

A `MultiSurface` is a geometry collection composed of surface elements. `MultiSurface` is a non-instantiable class. Its only instantiable subclass is `MultiPolygon`.

### `MultiSurface` Assertions

- Two `MultiSurface` surfaces have no interiors that intersect.
- Two `MultiSurface` elements have boundaries that intersect at most at a finite number of points.

## 7.2.13 Class `MultiPolygon`

A `MultiPolygon` is a `MultiSurface` object composed of `Polygon` elements.

### `MultiPolygon` Examples

- On a region map, a `MultiPolygon` could represent a system of lakes.

### `MultiPolygon` Assertions

- A `MultiPolygon` has no two `Polygon` elements with interiors that intersect.
- A `MultiPolygon` has no two `Polygon` elements that cross (crossing is also forbidden by the previous assertion), or that touch at an infinite number of points.
- A `MultiPolygon` may not have cut lines, spikes, or punctures. A `MultiPolygon` is a regular, closed point set.
- A `MultiPolygon` that has more than one `Polygon` has an interior that is not connected. The number of connected components of the interior of a `MultiPolygon` is equal to the number of `Polygon` values in the `MultiPolygon`.

### `MultiPolygon` Properties

- A `MultiPolygon` is a two-dimensional geometry.
- A `MultiPolygon` boundary is a set of closed curves (`LineString` values) corresponding to the boundaries of its `Polygon` elements.
- Each `Curve` in the boundary of the `MultiPolygon` is in the boundary of exactly one `Polygon` element.
- Every `Curve` in the boundary of a `Polygon` element is in the boundary of the `MultiPolygon`.

## 7.3 Supported Spatial Data Formats

This section describes the standard spatial data formats that are used to represent geometry objects in queries. They are:

- Well-Known Text (WKT) format
- Well-Known Binary (WKB) format

Internally, MySQL stores geometry values in a format that is not identical to either WKT or WKB format.

### 7.3.1 Well-Known Text (WKT) Format

The Well-Known Text (WKT) representation of geometry is designed to exchange geometry data in ASCII form.

Examples of WKT representations of geometry objects are:

- A `Point`:

  ```
  POINT(15 20)
  ```

  Note that point coordinates are specified with no separating comma.
- A `LineString` with four points:

  ```
  LINESTRING(0 0, 10 10, 20 25, 50 60)
  ```

  Note that point coordinate pairs are separated by commas.
- A `Polygon` with one exterior ring and one interior ring:

  ```
  POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))
  ```
- A `MultiPoint` with three `Point` values:

  ```
  MULTIPOINT(0 0, 20 20, 60 60)
  ```
- A `MultiLineString` with two `LineString` values:

  ```
  MULTILINESTRING((10 10, 20 20), (15 15, 30 15))
  ```

- A `MultiPolygon` with two `Polygon` values:

  ```
  MULTIPOLYGON(((0 0,10 0,10 10,0 10,0 0)),((5 5,7 5,7 7,5 7, 5 5)))
  ```

- A `GeometryCollection` consisting of two `Point` values and one `LineString`:

  ```
  GEOMETRYCOLLECTION(POINT(10 10), POINT(30 30), LINESTRING(15 15, 20 20))
  ```

A Backus-Naur grammar that specifies the formal production rules for writing WKT values can be found in the OGC specification document referenced near the beginning of this chapter.

### 7.3.2 Well-Known Binary (WKB) Format

The Well-Known Binary (WKB) representation for geometric values is defined by the OpenGIS specifications. It is also defined in the ISO "SQL/MM Part 3: Spatial" standard.

WKB is used to exchange geometry data as binary streams represented by `BLOB` values containing geometric WKB information.

WKB uses 1-byte unsigned integers, 4-byte unsigned integers, and 8-byte double-precision numbers (IEEE 754 format). A byte is 8 bits.

For example, a WKB value that corresponds to `POINT(1 1)` consists of this sequence of 21 bytes (each represented here by two hex digits):

```
0101000000000000000000F03F000000000000F03F
```

The sequence may be broken down into these components:

```
Byte order : 01
WKB type   : 01000000
X          : 000000000000F03F
Y          : 000000000000F03F
```

Component representation is as follows:

- The byte order may be either 0 or 1 to indicate little-endian or big-endian storage. The little-endian and big-endian byte orders are also known as Network Data Representation (NDR) and External Data Representation (XDR), respectively.
- The WKB type is a code that indicates the geometry type. Values from 1 through 7 indicate `Point`, `LineString`, `Polygon`, `MultiPoint`, `MultiLineString`, `MultiPolygon`, and `GeometryCollection`.
- A `Point` value has X and Y coordinates, each represented as a double-precision value.

WKB values for more complex geometry values are represented by more complex data structures, as detailed in the OpenGIS specification.

## 7.4 Creating a Spatially Enabled MySQL Database

This section describes the data types you can use for representing spatial data in MySQL, and the functions available for creating and retrieving spatial values.

### 7.4.1 MySQL Spatial Data Types

MySQL has data types that correspond to OpenGIS classes. Some of these types hold single geometry values:

- `GEOMETRY`
- `POINT`
- `LINESTRING`
- `POLYGON`

`GEOMETRY` can store geometry values of any type. The other single-value types, `POINT` and `LINESTRING` and `POLYGON`, restrict their values to a particular geometry type.

The other data types hold collections of values:

- `MULTIPOINT`
- `MULTILINESTRING`
- `MULTIPOLYGON`
- `GEOMETRYCOLLECTION`

`GEOMETRYCOLLECTION` can store a collection of objects of any type. The other collection types, `MULTIPOINT` and `MULTILINESTRING` and `MULTIPOLYGON` and `GEOMETRYCOLLECTION`, restrict collection members to those having a particular geometry type.

### 7.4.2 Creating Spatial Values

This section describes how to create spatial values using Well-Known Text and Well-Known Binary functions that are defined in the OpenGIS standard, and using MySQL-specific functions.

#### 7.4.2.1 Creating Geometry Values Using WKT Functions

MySQL provides a number of functions that take as input parameters a Well-Known Text representation and, optionally, a spatial reference system identifier (SRID). They return the corresponding geometry.

`GeomFromText()` accepts a WKT of any geometry type as its first argument. An implementation also provides type-specific construction functions for construction of geometry values of each geometry type.

- `GeomCollFromText(`*`wkt`*`[,`*`srid`*`])`, `GeometryCollectionFromText(`*`wkt`*`[,`*`srid`*`])`

  Constructs a `GEOMETRYCOLLECTION` value using its WKT representation and SRID.
- `GeomFromText(`*`wkt`*`[,`*`srid`*`])`, `GeometryFromText(`*`wkt`*`[,`*`srid`*`])`

  Constructs a geometry value of any type using its WKT representation and SRID.
- `LineFromText(`*`wkt`*`[,`*`srid`*`])`, `LineStringFromText(`*`wkt`*`[,`*`srid`*`])`

  Constructs a `LINESTRING` value using its WKT representation and SRID.
- `MLineFromText(`*`wkt`*`[,`*`srid`*`])`, `MultiLineStringFromText(`*`wkt`*`[,`*`srid`*`])`

  Constructs a `MULTILINESTRING` value using its WKT representation and SRID.
- `MPointFromText(`*`wkt`*`[,`*`srid`*`])`, `MultiPointFromText(`*`wkt`*`[,`*`srid`*`])`

  Constructs a `MULTIPOINT` value using its WKT representation and SRID.
- `MPolyFromText(`*`wkt`*`[,`*`srid`*`])`, `MultiPolygonFromText(`*`wkt`*`[,`*`srid`*`])`

  Constructs a `MULTIPOLYGON` value using its WKT representation and SRID.
- `PointFromText(`*`wkt`*`[,`*`srid`*`])`

  Constructs a `POINT` value using its WKT representation and SRID.
- `PolyFromText(`*`wkt`*`[,`*`srid`*`]), PolygonFromText(`*`wkt`*`[,`*`srid`*`])`

  Constructs a `POLYGON` value using its WKT representation and SRID.

The OpenGIS specification also describes optional functions for constructing `Polygon` or `MultiPolygon` values based on the WKT representation of a collection of rings or closed `LineString` values. These values may intersect. MySQL does not implement these functions:

- `BdMPolyFromText(`*`wkt`*`,`*`srid`*`)`

  Constructs a `MultiPolygon` value from a `MultiLineString` value in WKT format containing an arbitrary collection of closed `LineString` values.
- `BdPolyFromText(`*`wkt`*`,`*`srid`*`)`

  Constructs a `Polygon` value from a `MultiLineString` value in WKT format containing an arbitrary collection of closed `LineString` values.

### 7.4.2.2 Creating Geometry Values Using WKB Functions

MySQL provides a number of functions that take as input parameters a `BLOB` containing a Well-Known Binary representation and, optionally, a spatial reference system identifier (SRID). They return the corresponding geometry.

`GeomFromWKT()` accepts a WKB of any geometry type as its first argument. An implementation also provides type-specific construction functions for construction of geometry values of each geometry type.

- `GeomCollFromWKB(`*`wkb`*`[,`*`srid`*`])`, `GeometryCollectionFromWKB(`*`wkt`*`[,`*`srid`*`])`

  Constructs a `GEOMETRYCOLLECTION` value using its WKB representation and SRID.
- `GeomFromWKB(`*`wkb`*`[,`*`srid`*`])`, `GeometryFromWKB(`*`wkt`*`[,`*`srid`*`])`

  Constructs a geometry value of any type using its WKB representation and SRID.
- `LineFromWKB(`*`wkb`*`[,`*`srid`*`])`, `LineStringFromWKB(`*`wkb`*`[,`*`srid`*`])`

  Constructs a `LINESTRING` value using its WKB representation and SRID.
- `MLineFromWKB(`*`wkb`*`[,`*`srid`*`])`, `MultiLineStringFromWKB(`*`wkb`*`[,`*`srid`*`])`

  Constructs a `MULTILINESTRING` value using its WKB representation and SRID.
- `MPointFromWKB(`*`wkb`*`[,`*`srid`*`])`, `MultiPointFromWKB(`*`wkb`*`[,`*`srid`*`])`

  Constructs a `MULTIPOINT` value using its WKB representation and SRID.
- `MPolyFromWKB(`*`wkb`*`[,`*`srid`*`])`, `MultiPolygonFromWKB(`*`wkb`*`[,`*`srid`*`])`

  Constructs a `MULTIPOLYGON` value using its WKB representation and SRID.
- `PointFromWKB(`*`wkb`*`[,`*`srid`*`])`

  Constructs a `POINT` value using its WKB representation and SRID.
- `PolyFromWKB(`*`wkb`*`[,`*`srid`*`])`, `PolygonFromWKB(`*`wkb`*`[,`*`srid`*`])`

  Constructs a `POLYGON` value using its WKB representation and SRID.

The OpenGIS specification also describes optional functions for constructing `Polygon` or `MultiPolygon` values based on the WKB representation of a collection of rings or closed `LineString` values. These values may intersect. MySQL does not implement these functions:

- `BdMPolyFromWKB(`*`wkb`*`,`*`srid`*`)`

  Constructs a `MultiPolygon` value from a `MultiLineString` value in WKB format containing an arbitrary collection of closed `LineString` values.
- `BdPolyFromWKB(`*`wkb`*`,`*`srid`*`)`

  Constructs a `Polygon` value from a `MultiLineString` value in WKB format containing an arbitrary collection of closed `LineString` values.

### 7.4.2.3 Creating Geometry Values Using MySQL-Specific Functions

**Note:** MySQL does not implement the functions listed in this section.

MySQL provides a set of useful functions for creating geometry WKB representations. The functions described in this section are MySQL extensions to the OpenGIS specifications. The results of these functions are `BLOB` values containing WKB representations of geometry values with no SRID. The results of these functions can be substituted as the first argument for any function in the `GeomFromWKB()` function family.

- `GeometryCollection(`*`g1`*`,`*`g2`*`,...)`

  Constructs a WKB `GeometryCollection`. If any argument is not a well-formed WKB representation of a geometry, the return value is `NULL`.

- `LineString(`*`pt1`*`,`*`pt2`*`,...)`

  Constructs a WKB `LineString` value from a number of WKB `Point` arguments. If any argument is not a WKB `Point`, the return value is `NULL`. If the number of `Point` arguments is less than two, the return value is `NULL`.

- `MultiLineString(`*`ls1`*`,`*`ls2`*`,...)`

  Constructs a WKB `MultiLineString` value using WKB `LineString` arguments. If any argument is not a WKB `LineString`, the return value is `NULL`.

- `MultiPoint(`*`pt1`*`,`*`pt2`*`,...)`

  Constructs a WKB `MultiPoint` value using WKB `Point` arguments. If any argument is not a WKB `Point`, the return value is `NULL`.

- `MultiPolygon(`*`poly1`*`,`*`poly2`*`,...)`

  Constructs a WKB `MultiPolygon` value from a set of WKB `Polygon` arguments. If any argument is not a WKB `Polygon`, the return value is `NULL`.

- `Point(`*`x`*`,`*`y`*`)`

  Constructs a WKB `Point` using its coordinates.

- `Polygon(`*`ls1`*`,`*`ls2`*`,...)`

  Constructs a WKB `Polygon` value from a number of WKB `LineString` arguments. If any argument does not represent the WKB of a `LinearRing` (that is, not a closed and simple `LineString`) the return value is `NULL`.

## 7.4.3 Creating Spatial Columns

MySQL provides a standard way of creating spatial columns for geometry types, for example, with `CREATE TABLE` or `ALTER TABLE`. Currently, spatial columns are supported only for `MyISAM` tables.

- Use the `CREATE TABLE` statement to create a table with a spatial column:

```
mysql> CREATE TABLE geom (g GEOMETRY);
Query OK, 0 rows affected (0.02 sec)
```

- Use the ALTER TABLE statement to add or drop a spatial column to or from an existing table:

  ```
  mysql> ALTER TABLE geom ADD pt POINT;
  Query OK, 0 rows affected (0.00 sec)
  Records: 0  Duplicates: 0  Warnings: 0
  mysql> ALTER TABLE geom DROP pt;
  Query OK, 0 rows affected (0.00 sec)
  Records: 0  Duplicates: 0  Warnings: 0
  ```

## 7.4.4 Populating Spatial Columns

After you have created spatial columns, you can populate them with spatial data.

Values should be stored in internal geometry format, but you can convert them to that format from either Well-Known Text (WKT) or Well-Known Binary (WKB) format. The following examples demonstrate how to insert geometry values into a table by converting WKT values into internal geometry format.

You can perform the conversion directly in the INSERT statement:

```
INSERT INTO geom VALUES (GeomFromText('POINT(1 1)'));

SET @g = 'POINT(1 1)';
INSERT INTO geom VALUES (GeomFromText(@g));
```

Or you can perform the conversion prior to the INSERT:

```
SET @g = GeomFromText('POINT(1 1)');
INSERT INTO geom VALUES (@g);
```

The following examples insert more complex geometries into the table:

```
SET @g = 'LINESTRING(0 0,1 1,2 2)';
INSERT INTO geom VALUES (GeomFromText(@g));

SET @g = 'POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))';
INSERT INTO geom VALUES (GeomFromText(@g));

SET @g =
'GEOMETRYCOLLECTION(POINT(1 1),LINESTRING(0 0,1 1,2 2,3 3,4 4))';
INSERT INTO geom VALUES (GeomFromText(@g));
```

The preceding examples all use GeomFromText() to create geometry values. You can also use type-specific functions:

```
SET @g = 'POINT(1 1)';
INSERT INTO geom VALUES (PointFromText(@g));
```

```
SET @g = 'LINESTRING(0 0,1 1,2 2)';
INSERT INTO geom VALUES (LineStringFromText(@g));

SET @g = 'POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))';
INSERT INTO geom VALUES (PolygonFromText(@g));

SET @g =
GEOMETRYCOLLECTION(POINT(1 1),LINESTRING(0 0,1 1,2 2,3 3,4 4))';
INSERT INTO geom VALUES (GeomCollFromText(@g));
```

Note that if a client application program wants to use WKB representations of geometry values, it is responsible for sending correctly formed WKB in queries to the server. However, there are several ways of satisfying this requirement. For example:

- Inserting a `POINT(1 1)` value with hex literal syntax:

  ```
  mysql> INSERT INTO geom VALUES
      -> (GeomFromWKB(0x0101000000000000000000F03F000000000000F03F));
  ```

- An ODBC application can send a WKB representation, binding it to a placeholder using an argument of `BLOB` type:

  ```
  INSERT INTO geom VALUES (GeomFromWKB(?))
  ```

  Other programming interfaces may support a similar placeholder mechanism.
- In a C program, you can escape a binary value using `mysql_real_escape_string()` and include the result in a query string that is sent to the server.

## 7.4.5 Fetching Spatial Data

Geometry values stored in a table can be fetched in internal format. You can also convert them into WKT or WKB format.

### 7.4.5.1 Fetching Spatial Data in Internal Format

Fetching geometry values using internal format can be useful in table-to-table transfers:

```
CREATE TABLE geom2 (g GEOMETRY) SELECT g FROM geom;
```

### 7.4.5.2 Fetching Spatial Data in WKT Format

The `AsText()` function converts a geometry from internal format into a WKT string.

```
mysql> SELECT AsText(g) FROM geom;
+-------------------------+
| AsText(p1)              |
+-------------------------+
| POINT(1 1)              |
| LINESTRING(0 0,1 1,2 2) |
+-------------------------+
```

### 7.4.5.3 Fetching Spatial Data in WKB Format

The `AsBinary()` function converts a geometry from internal format into a `BLOB` containing the WKB value.

```
SELECT AsBinary(g) FROM geom;
```

## 7.5 Analyzing Spatial Information

After populating spatial columns with values, you are ready to query and analyze them. MySQL provides a set of functions to perform various operations on spatial data. These functions can be grouped into four major categories according to the type of operation they perform:

- Functions that convert geometries between various formats
- Functions that provide access to qualitative or quantitative properties of a geometry
- Functions that describe relations between two geometries
- Functions that create new geometries from existing ones

Spatial analysis functions can be used in many contexts, such as:

- Any interactive SQL program, such as `mysql` or `MySQLCC`
- Application programs written in any language that supports a MySQL client API

### 7.5.1 Geometry Format Conversion Functions

MySQL supports the following functions for converting geometry values between internal format and either WKT or WKB format:

- `AsBinary(`*g*`)`

  Converts a value in internal geometry format to its WKB representation and returns the binary result.

- `AsText(`*g*`)`

  Converts a value in internal geometry format to its WKT representation and returns the string result.

  ```
  mysql> SET @g = 'LineString(1 1,2 2,3 3)';
  mysql> SELECT AsText(GeomFromText(@g));
  +--------------------------+
  | AsText(GeomFromText(@G)) |
  +--------------------------+
  | LINESTRING(1 1,2 2,3 3)  |
  +--------------------------+
  ```

- `GeomFromText(wkt[,srid])`

  Converts a string value from its WKT representation into internal geometry format and returns the result. A number of type-specific functions are also supported, such as `PointFromText()` and `LineFromText()`; see Section 7.4.2.1, "Creating Geometry Values Using WKT Functions."

- `GeomFromWKB(wkb[,srid])`

  Converts a binary value from its WKB representation into internal geometry format and returns the result. A number of type-specific functions are also supported, such as `PointFromWKB()` and `LineFromWKB()`; see Section 7.4.2.2, "Creating Geometry Values Using WKB Functions."

### 7.5.2 `Geometry` Functions

Each function that belongs to this group takes a geometry value as its argument and returns some qualitative or quantitative property of the geometry. Some functions restrict their argument type. Such functions return `NULL` if the argument is of an incorrect geometry type. For example, `Area()` returns `NULL` if the object type is neither `Polygon` nor `MultiPolygon`.

#### 7.5.2.1 General Geometry Functions

The functions listed in this section do not restrict their argument and accept a geometry value of any type.

- `Dimension(g)`

  Returns the inherent dimension of the geometry value *g*. The result can be –1, 0, 1, or 2. (The meaning of these values is given in Section 7.2.2, "Class `Geometry`.")

```
mysql> SELECT Dimension(GeomFromText('LineString(1 1,2 2)'));
+------------------------------------------------+
| Dimension(GeomFromText('LineString(1 1,2 2)')) |
+------------------------------------------------+
|                                              1 |
+------------------------------------------------+
```

- `Envelope(g)`

  Returns the Minimum Bounding Rectangle (MBR) for the geometry value *g*. The result is returned as a `Polygon` value.

```
mysql> SELECT AsText(Envelope(GeomFromText('LineString(1 1,2 2)')));
+-------------------------------------------------------+
| AsText(Envelope(GeomFromText('LineString(1 1,2 2)'))) |
+-------------------------------------------------------+
| POLYGON((1 1,2 1,2 2,1 2,1 1))                        |
+-------------------------------------------------------+
```

The polygon is defined by the corner points of the bounding box:

```
POLYGON((MINX MINY, MAXX MINY, MAXX MAXY, MINX MAXY, MINX MINY))
```

- `GeometryType(g)`

  Returns as a string the name of the geometry type of which the geometry instance *g* is a member. The name will correspond to one of the instantiable `Geometry` subclasses.

  ```
  mysql> SELECT GeometryType(GeomFromText('POINT(1 1)'));
  +------------------------------------------+
  | GeometryType(GeomFromText('POINT(1 1)')) |
  +------------------------------------------+
  | POINT                                    |
  +------------------------------------------+
  ```

- `SRID(g)`

  Returns an integer indicating the Spatial Reference System ID for the geometry value *g*.

  ```
  mysql> SELECT SRID(GeomFromText('LineString(1 1,2 2)',101));
  +-----------------------------------------------+
  | SRID(GeomFromText('LineString(1 1,2 2)',101)) |
  +-----------------------------------------------+
  |                                           101 |
  +-----------------------------------------------+
  ```

The OpenGIS specification also defines the following functions, which MySQL does not implement:

- `Boundary(g)`

  Returns a geometry that is the closure of the combinatorial boundary of the geometry value *g*.

- `IsEmpty(g)`

  Returns 1 if the geometry value *g* is the empty geometry, 0 if it is not empty, and –1 if the argument is `NULL`. If the geometry is empty, it represents the empty point set.

- `IsSimple(g)`

  Currently, this function is a placeholder and should not be used. If implemented, its behavior will be as described in the next paragraph.

  Returns 1 if the geometry value *g* has no anomalous geometric points, such as self-intersection or self-tangency. `IsSimple()` returns 0 if the argument is not simple, and –1 if it is `NULL`.

  The description of each instantiable geometric class given earlier in the chapter includes the specific conditions that cause an instance of that class to be classified as not simple.

### 7.5.2.2 `Point` Functions

A `Point` consists of X and Y coordinates, which may be obtained using the following functions:

- `X(p)`

  Returns the X-coordinate value for the point *p* as a double-precision number.

```
mysql> SELECT X(GeomFromText('Point(56.7 53.34)'));
+--------------------------------------+
| X(GeomFromText('Point(56.7 53.34)')) |
+--------------------------------------+
|                                 56.7 |
+--------------------------------------+
```

- `Y(p)`

  Returns the Y-coordinate value for the point *p* as a double-precision number.

```
mysql> SELECT Y(GeomFromText('Point(56.7 53.34)'));
+--------------------------------------+
| Y(GeomFromText('Point(56.7 53.34)')) |
+--------------------------------------+
|                                53.34 |
+--------------------------------------+
```

### 7.5.2.3 `LineString` Functions

A `LineString` consists of `Point` values. You can extract particular points of a `LineString`, count the number of points that it contains, or obtain its length.

- `EndPoint(ls)`

  Returns the `Point` that is the end point of the `LineString` value *ls*.

```
mysql> SET @ls = 'LineString(1 1,2 2,3 3)';
mysql> SELECT AsText(EndPoint(GeomFromText(@ls)));
+-------------------------------------+
| AsText(EndPoint(GeomFromText(@ls))) |
+-------------------------------------+
| POINT(3 3)                          |
+-------------------------------------+
```

- `GLength(`*`ls`*`)`

  Returns as a double-precision number the length of the `LineString` value *`ls`* in its associated spatial reference.

```
mysql> SET @ls = 'LineString(1 1,2 2,3 3)';
mysql> SELECT GLength(GeomFromText(@ls));
+----------------------------+
| GLength(GeomFromText(@ls)) |
+----------------------------+
|            2.8284271247462 |
+----------------------------+
```

- `IsClosed(`*`ls`*`)`

  Returns 1 if the `LineString` value *`ls`* is closed (that is, its `StartPoint()` and `EndPoint()` values are the same). Returns 0 if *`ls`* is not closed, and –1 if it is `NULL`.

```
mysql> SET @ls = 'LineString(1 1,2 2,3 3)';
mysql> SELECT IsClosed(GeomFromText(@ls));
+-----------------------------+
| IsClosed(GeomFromText(@ls)) |
+-----------------------------+
|                           0 |
+-----------------------------+
```

- `NumPoints(`*`ls`*`)`

  Returns the number of points in the `LineString` value *`ls`*.

```
mysql> SET @ls = 'LineString(1 1,2 2,3 3)';
mysql> SELECT NumPoints(GeomFromText(@ls));
+------------------------------+
| NumPoints(GeomFromText(@ls)) |
+------------------------------+
|                            3 |
+------------------------------+
```

- `PointN(`*`ls`*`,`*`n`*`)`

  Returns the *n*-th point in the `Linestring` value *`ls`*. Point numbers begin at 1.

```
mysql> SET @ls = 'LineString(1 1,2 2,3 3)';
mysql> SELECT AsText(PointN(GeomFromText(@ls),2));
+-------------------------------------+
| AsText(PointN(GeomFromText(@ls),2)) |
+-------------------------------------+
| POINT(2 2)                          |
+-------------------------------------+
```

- `StartPoint(ls)`

  Returns the `Point` that is the start point of the `LineString` value *`ls`*.

```
mysql> SET @ls = 'LineString(1 1,2 2,3 3)';
mysql> SELECT AsText(StartPoint(GeomFromText(@ls)));
+---------------------------------------+
| AsText(StartPoint(GeomFromText(@ls))) |
+---------------------------------------+
| POINT(1 1)                            |
+---------------------------------------+
```

The OpenGIS specification also defines the following function, which MySQL does not implement:

- `IsRing(ls)`

  Returns 1 if the `LineString` value *`ls`* is closed (that is, its `StartPoint()` and `EndPoint()` values are the same) and is simple (does not pass through the same point more than once). Returns 0 if *`ls`* is not a ring, and –1 if it is `NULL`.

### 7.5.2.4 `MultiLineString` Functions

- `GLength(mls)`

  Returns as a double-precision number the length of the `MultiLineString` value *`mls`*. The length of *`mls`* is equal to the sum of the lengths of its elements.

```
mysql> SET @mls = 'MultiLineString((1 1,2 2,3 3),(4 4,5 5))';
mysql> SELECT GLength(GeomFromText(@mls));
+-----------------------------+
| GLength(GeomFromText(@mls)) |
+-----------------------------+
|             4.2426406871193 |
+-----------------------------+
```

- `IsClosed(mls)`

  Returns 1 if the `MultiLineString` value *`mls`* is closed (that is, the `StartPoint()` and `EndPoint()` values are the same for each `LineString` in *`mls`*). Returns 0 if *`mls`* is not closed, and –1 if it is `NULL`.

```
mysql> SET @mls = 'MultiLineString((1 1,2 2,3 3),(4 4,5 5))';
mysql> SELECT IsClosed(GeomFromText(@mls));
+------------------------------+
| IsClosed(GeomFromText(@mls)) |
+------------------------------+
|                            0 |
+------------------------------+
```

#### 7.5.2.5 `Polygon` Functions

- `Area(`*`poly`*`)`

  Returns as a double-precision number the area of the `Polygon` value *`poly`*, as measured in its spatial reference system.

```
mysql> SET @poly = 'Polygon((0 0,0 3,3 0,0 0),(1 1,1 2,2 1,1 1))';
mysql> SELECT Area(GeomFromText(@poly));
+---------------------------+
| Area(GeomFromText(@poly)) |
+---------------------------+
|                         4 |
+---------------------------+
```

- `ExteriorRing(`*`poly`*`)`

  Returns the exterior ring of the `Polygon` value *`poly`* as a `LineString`.

```
mysql> SET @poly =
    -> 'Polygon((0 0,0 3,3 3,3 0,0 0),(1 1,1 2,2 2,2 1,1 1))';
mysql> SELECT AsText(ExteriorRing(GeomFromText(@poly)));
+-------------------------------------------+
| AsText(ExteriorRing(GeomFromText(@poly))) |
+-------------------------------------------+
| LINESTRING(0 0,0 3,3 3,3 0,0 0)           |
+-------------------------------------------+
```

- `InteriorRingN(`*`poly`*`,`*`n`*`)`

  Returns the *n*-th interior ring for the `Polygon` value *`poly`* as a `LineString`. Ring numbers begin at 1.

```
mysql> SET @poly =
    -> 'Polygon((0 0,0 3,3 3,3 0,0 0),(1 1,1 2,2 2,2 1,1 1))';
mysql> SELECT AsText(InteriorRingN(GeomFromText(@poly),1));
+----------------------------------------------+
| AsText(InteriorRingN(GeomFromText(@poly),1)) |
+----------------------------------------------+
| LINESTRING(1 1,1 2,2 2,2 1,1 1)              |
+----------------------------------------------+
```

- `NumInteriorRings(`*`poly`*`)`

  Returns the number of interior rings in the `Polygon` value *`poly`*.

```
mysql> SET @poly =
    -> 'Polygon((0 0,0 3,3 3,3 0,0 0),(1 1,1 2,2 2,2 1,1 1))';
mysql> SELECT NumInteriorRings(GeomFromText(@poly));
+---------------------------------------+
| NumInteriorRings(GeomFromText(@poly)) |
+---------------------------------------+
|                                     1 |
+---------------------------------------+
```

### 7.5.2.6 `MultiPolygon` Functions

- `Area(`*`mpoly`*`)`

  Returns as a double-precision number the area of the `MultiPolygon` value *`mpoly`*, as measured in its spatial reference system.

```
mysql> SET @mpoly =
    -> 'MultiPolygon(((0 0,0 3,3 3,3 0,0 0),(1 1,1 2,2 2,2 1,1 1)))';
mysql> SELECT Area(GeomFromText(@mpoly));
+----------------------------+
| Area(GeomFromText(@mpoly)) |
+----------------------------+
|                          8 |
+----------------------------+
```

The OpenGIS specification also defines the following functions, which MySQL does not implement:

- `Centroid(`*`mpoly`*`)`

  Returns the mathematical centroid for the `MultiPolygon` value *`mpoly`* as a `Point`. The result is not guaranteed to be on the `MultiPolygon`.
- `PointOnSurface(`*`mpoly`*`)`

  Returns a `Point` value that is guaranteed to be on the `MultiPolygon` value *`mpoly`*.

### 7.5.2.7 `GeometryCollection` Functions

- `GeometryN(`*`gc`*`,`*`n`*`)`

  Returns the *n*-th geometry in the `GeometryCollection` value *`gc`*. Geometry numbers begin at 1.

```
mysql> SET @gc = 'GeometryCollection(Point(1 1),LineString(2 2, 3 3))';
mysql> SELECT AsText(GeometryN(GeomFromText(@gc),1));
+----------------------------------------+
| AsText(GeometryN(GeomFromText(@gc),1)) |
+----------------------------------------+
| POINT(1 1)                             |
+----------------------------------------+
```

- `NumGeometries(gc)`

  Returns the number of geometries in the `GeometryCollection` value *gc*.

  ```
  mysql> SET @gc = 'GeometryCollection(Point(1 1),LineString(2 2, 3 3))';
  mysql> SELECT NumGeometries(GeomFromText(@gc));
  +----------------------------------+
  | NumGeometries(GeomFromText(@gc)) |
  +----------------------------------+
  |                                2 |
  +----------------------------------+
  ```

## 7.5.3 Functions That Create New Geometries from Existing Ones

### 7.5.3.1 Geometry Functions That Produce New Geometries

In Section 7.5.2, "`Geometry` Functions," we've already discussed some functions that can construct new geometries from the existing ones:

- `Envelope(g)`
- `StartPoint(ls)`
- `EndPoint(ls)`
- `PointN(ls,n)`
- `ExteriorRing(poly)`
- `InteriorRingN(poly,n)`
- `GeometryN(gc,n)`

### 7.5.3.2 Spatial Operators

OpenGIS proposes a number of other functions that can produce geometries. They are designed to implement spatial operators.

These functions are not implemented in MySQL. They may appear in future releases.

- `Buffer(g,d)`

  Returns a geometry that represents all points whose distance from the geometry value *g* is less than or equal to a distance of *d*.

- `ConvexHull(g)`

  Returns a geometry that represents the convex hull of the geometry value *g*.

- `Difference(g1,g2)`

  Returns a geometry that represents the point set difference of the geometry value *g1* with *g2*.
- `Intersection(g1,g2)`

  Returns a geometry that represents the point set intersection of the geometry values *g1* with *g2*.
- `SymDifference(g1,g2)`

  Returns a geometry that represents the point set symmetric difference of the geometry value *g1* with *g2*.
- `Union(g1,g2)`

  Returns a geometry that represents the point set union of the geometry values *g1* and *g2*.

## 7.5.4 Functions for Testing Spatial Relations Between Geometric Objects

The functions described in these sections take two geometries as input parameters and return a qualitative or quantitative relation between them.

## 7.5.5 Relations on Geometry Minimal Bounding Rectangles (MBRs)

MySQL provides some functions that can test relations between minimal bounding rectangles of two geometries *g1* and *g2*. They include:

- `MBRContains(g1,g2)`

  Returns 1 or 0 to indicate whether or not the Minimum Bounding Rectangle of *g1* contains the Minimum Bounding Rectangle of *g2*.

```
mysql> SET @g1 = GeomFromText('Polygon((0 0,0 3,3 3,3 0,0 0))');
mysql> SET @g2 = GeomFromText('Point(1 1)');
mysql> SELECT MBRContains(@g1,@g2), MBRContains(@g2,@g1);
----------------------+----------------------+
| MBRContains(@g1,@g2) | MBRContains(@g2,@g1) |
+----------------------+----------------------+
|                    1 |                    0 |
+----------------------+----------------------+
```

- `MBRDisjoint(g1,g2)`

  Returns 1 or 0 to indicate whether or not the Minimum Bounding Rectangles of the two geometries *g1* and *g2* are disjointed (do not intersect).

- `MBREqual(g1,g2)`

  Returns 1 or 0 to indicate whether or not the Minimum Bounding Rectangles of the two geometries *g1* and *g2* are the same.

- `MBRIntersects(g1,g2)`

  Returns 1 or 0 to indicate whether or not the Minimum Bounding Rectangles of the two geometries *g1* and *g2* intersect.

- `MBROverlaps(g1,g2)`

  Returns 1 or 0 to indicate whether or not the Minimum Bounding Rectangles of the two geometries *g1* and *g2* overlap.

- `MBRTouches(g1,g2)`

  Returns 1 or 0 to indicate whether or not the Minimum Bounding Rectangles of the two geometries *g1* and *g2* touch.

- `MBRWithin(g1,g2)`

  Returns 1 or 0 to indicate whether or not the Minimum Bounding Rectangle of *g1* is within the Minimum Bounding Rectangle of *g2*.

```
mysql> SET @g1 = GeomFromText('Polygon((0 0,0 3,3 3,3 0,0 0))');
mysql> SET @g2 = GeomFromText('Polygon((0 0,0 5,5 5,5 0,0 0))');
mysql> SELECT MBRWithin(@g1,@g2), MBRWithin(@g2,@g1);
+--------------------+--------------------+
| MBRWithin(@g1,@g2) | MBRWithin(@g2,@g1) |
+--------------------+--------------------+
|                  1 |                  0 |
+--------------------+--------------------+
```

### 7.5.6 Functions That Test Spatial Relationships Between Geometries

The OpenGIS specification defines the following functions. Currently, MySQL does not implement them according to the specification. Those that are implemented return the same result as the corresponding MBR-based functions. This includes functions in the following list other than `Distance()` and `Related()`.

These functions may be implemented in future releases with full support for spatial analysis, not just MBR-based support.

The functions operate on two geometry values *g1* and *g2*.

- `Contains(g1,g2)`

  Returns 1 or 0 to indicate whether or not *g1* completely contains *g2*.

- `Crosses(g1,g2)`

  Returns 1 if *g1* spatially crosses *g2*. Returns `NULL` if *g1* is a `Polygon` or a `MultiPolygon`, or if *g2* is a `Point` or a `MultiPoint`. Otherwise, returns 0.

  The term *spatially crosses* denotes a spatial relation between two given geometries that has the following properties:

  - The two geometries intersect
  - Their intersection results in a geometry that has a dimension that is one less than the maximum dimension of the two given geometries
  - Their intersection is not equal to either of the two given geometries

- `Disjoint(g1,g2)`

  Returns 1 or 0 to indicate whether or not *g1* is spatially disjointed from (does not intersect) *g2*.

- `Distance(g1,g2)`

  Returns as a double-precision number the shortest distance between any two points in the two geometries.

- `Equals(g1,g2)`

  Returns 1 or 0 to indicate whether or not *g1* is spatially equal to *g2*.

- `Intersects(g1,g2)`

  Returns 1 or 0 to indicate whether or not *g1* spatially intersects *g2*.

- `Overlaps(g1,g2)`

  Returns 1 or 0 to indicate whether or not *g1* spatially overlaps *g2*. The term *spatially overlaps* is used if two geometries intersect and their intersection results in a geometry of the same dimension but not equal to either of the given geometries.

- `Related(g1,g2,pattern_matrix)`

  Returns 1 or 0 to indicate whether or not the spatial relationship specified by *pattern_matrix* exists between *g1* and *g2*. Returns –1 if the arguments are `NULL`. The pattern matrix is a string. Its specification will be noted here if this function is implemented.

- `Touches(g1,g2)`

  Returns 1 or 0 to indicate whether or not *g1* spatially touches *g2*. Two geometries *spatially touch* if the interiors of the geometries do not intersect, but the boundary of one of the geometries intersects either the boundary or the interior of the other.
- `Within(g1,g2)`

  Returns 1 or 0 to indicate whether or not *g1* is spatially within *g2*.

## 7.6 Optimizing Spatial Analysis

Search operations in non-spatial databases can be optimized using indexes. This is true for spatial databases as well. With the help of a great variety of multi-dimensional indexing methods that have already been designed, it is possible to optimize spatial searches. The most typical of these are:

- Point queries that search for all objects that contain a given point
- Region queries that search for all objects that overlap a given region

MySQL uses **R-Trees with quadratic splitting** to index spatial columns. A spatial index is built using the MBR of a geometry. For most geometries, the MBR is a minimum rectangle that surrounds the geometries. For a horizontal or a vertical linestring, the MBR is a rectangle degenerated into the linestring. For a point, the MBR is a rectangle degenerated into the point.

### 7.6.1 Creating Spatial Indexes

MySQL can create spatial indexes using syntax similar to that for creating regular indexes, but extended with the `SPATIAL` keyword. Spatial columns that are indexed currently must be declared `NOT NULL`. The following examples demonstrate how to create spatial indexes.

- With `CREATE TABLE`:

  ```
  mysql> CREATE TABLE geom (g GEOMETRY NOT NULL, SPATIAL INDEX(g));
  ```

- With `ALTER TABLE`:

  ```
  mysql> ALTER TABLE geom ADD SPATIAL INDEX(g);
  ```

- With `CREATE INDEX`:

  ```
  mysql> CREATE SPATIAL INDEX sp_index ON geom (g);
  ```

To drop spatial indexes, use `ALTER TABLE` or `DROP INDEX`:

- With `ALTER TABLE`:

  ```
  mysql> ALTER TABLE geom DROP INDEX g;
  ```

- With `DROP INDEX`:

  ```
  mysql> DROP INDEX sp_index ON geom;
  ```

Example: Suppose that a table `geom` contains more than 32,000 geometries, which are stored in the column `g` of type `GEOMETRY`. The table also has an `AUTO_INCREMENT` column `fid` for storing object ID values.

```
mysql> DESCRIBE geom;
+-------+----------+------+-----+---------+----------------+
| Field | Type     | Null | Key | Default | Extra          |
+-------+----------+------+-----+---------+----------------+
| fid   | int(11)  |      | PRI | NULL    | auto_increment |
| g     | geometry |      |     |         |                |
+-------+----------+------+-----+---------+----------------+
2 rows in set (0.00 sec)

mysql> SELECT COUNT(*) FROM geom;
+----------+
| count(*) |
+----------+
|    32376 |
+----------+
1 row in set (0.00 sec)
```

To add a spatial index on the column `g`, use this statement:

```
mysql> ALTER TABLE geom ADD SPATIAL INDEX(g);
Query OK, 32376 rows affected (4.05 sec)
Records: 32376  Duplicates: 0  Warnings: 0
```

## 7.6.2 Using a Spatial Index

The optimizer investigates whether available spatial indexes can be involved in the search for queries that use a function such as `MBRContains()` or `MBRWithin()` in the `WHERE` clause. For example, let's say we want to find all objects that are in the given rectangle:

```
mysql> SELECT fid,AsText(g) FROM geom WHERE
mysql> MBRContains(GeomFromText('Polygon((30000 15000,31000 15000,31000
                                         ➥16000,30000 16000,30000 15000))'),g);
+-----+---------------------------------------------------------------------------+
| fid | AsText(g)                                                                 |
+-----+---------------------------------------------------------------------------+
|  21 | LINESTRING(30350.4 15828.8,30350.6 15845,30333.8 15845,30333.8 15828.8)     |
|  22 | LINESTRING(30350.6 15871.4,30350.6 15887.8,30334 15887.8,30334 15871.4)     |
|  23 | LINESTRING(30350.6 15914.2,30350.6 15930.4,30334 15930.4,30334 15914.2)     |
|  24 | LINESTRING(30290.2 15823,30290.2 15839.4,30273.4 15839.4,30273.4 15823)     |
|  25 | LINESTRING(30291.4 15866.2,30291.6 15882.4,30274.8 15882.4,30274.8 15866.2) |
|  26 | LINESTRING(30291.6 15918.2,30291.6 15934.4,30275 15934.4,30275 15918.2)     |
| 249 | LINESTRING(30337.8 15938.6,30337.8 15946.8,30320.4 15946.8,30320.4 15938.4) |
|   1 | LINESTRING(30250.4 15129.2,30248.8 15138.4,30238.2 15136.4,30240 15127.2)   |
|   2 | LINESTRING(30220.2 15122.8,30217.2 15137.8,30207.6 15136,30210.4 15121)     |
|   3 | LINESTRING(30179 15114.4,30176.6 15129.4,30167 15128,30169 15113)           |
|   4 | LINESTRING(30155.2 15121.4,30140.4 15118.6,30142 15109,30157 15111.6)       |
|   5 | LINESTRING(30192.4 15085,30177.6 15082.2,30179.2 15072.4,30194.2 15075.2)   |
|   6 | LINESTRING(30244 15087,30229 15086.2,30229.4 15076.4,30244.6 15077)         |
|   7 | LINESTRING(30200.6 15059.4,30185.6 15058.6,30186 15048.8,30201.2 15049.4)   |
|  10 | LINESTRING(30179.6 15017.8,30181 15002.8,30190.8 15003.6,30189.6 15019)     |
|  11 | LINESTRING(30154.2 15000.4,30168.6 15004.8,30166 15014.2,30151.2 15009.8)   |
|  13 | LINESTRING(30105 15065.8,30108.4 15050.8,30118 15053,30114.6 15067.8)       |
| 154 | LINESTRING(30276.2 15143.8,30261.4 15141,30263 15131.4,30278 15134)         |
| 155 | LINESTRING(30269.8 15084,30269.4 15093.4,30258.6 15093,30259 15083.4)       |
| 157 | LINESTRING(30128.2 15011,30113.2 15010.2,30113.6 15000.4,30128.8 15001)     |
+-----+---------------------------------------------------------------------------+
20 rows in set (0.00 sec)
```

Now let's use `EXPLAIN` to check the way this query is executed (the `id` output column has been removed so the output better fits the page):

```
mysql> EXPLAIN SELECT fid,AsText(g) FROM geom WHERE
mysql> MBRContains(GeomFromText('Polygon((30000 15000,31000 15000,31000
                                       ➥16000,30000 16000,30000 15000))'),g);
+-------------+-------+-------+---------------+------+---------+------+------+-------------+
| select_type | table | type  | possible_keys | key  | key_len | ref  | rows | Extra       |
+-------------+-------+-------+---------------+------+---------+------+------+-------------+
| SIMPLE      | geom  | range | g             | g    |      32 | NULL |   50 | Using where |
+-------------+-------+-------+---------------+------+---------+------+------+-------------+
1 row in set (0.00 sec)
```

Now let's check what would happen without a spatial index:

```
mysql> EXPLAIN SELECT fid,AsText(g) FROM g IGNORE INDEX (g) WHERE
mysql> MBRContains(GeomFromText('Polygon((30000 15000,31000 15000,31000
                                          ➥16000,30000 16000,30000 15000))'),g);
+-------------+-------+------+---------------+------+---------+------+-------+-------------+
| select_type | table | type | possible_keys | key  | key_len | ref  | rows  | Extra       |
+-------------+-------+------+---------------+------+---------+------+-------+-------------+
| SIMPLE      | geom  | ALL  | NULL          | NULL |    NULL | NULL | 32376 | Using where |
+-------------+-------+------+---------------+------+---------+------+-------+-------------+
1 row in set (0.00 sec)
```

Let's execute the SELECT statement, ignoring the spatial key we have:

```
mysql> SELECT fid,AsText(g) FROM geom IGNORE INDEX (g) WHERE
mysql> MBRContains(GeomFromText('Polygon((30000 15000,31000 15000,31000
                                          ➥16000,30000 16000,30000 15000))'),g);
+-----+-----------------------------------------------------------------------+
| fid | AsText(g)                                                             |
+-----+-----------------------------------------------------------------------+
|   1 | LINESTRING(30250.4 15129.2,30248.8 15138.4,30238.2 15136.4,30240 15127.2)   |
|   2 | LINESTRING(30220.2 15122.8,30217.2 15137.8,30207.6 15136,30210.4 15121)     |
|   3 | LINESTRING(30179 15114.4,30176.6 15129.4,30167 15128,30169 15113)           |
|   4 | LINESTRING(30155.2 15121.4,30140.4 15118.6,30142 15109,30157 15111.6)       |
|   5 | LINESTRING(30192.4 15085,30177.6 15082.2,30179.2 15072.4,30194.2 15075.2)   |
|   6 | LINESTRING(30244 15087,30229 15086.2,30229.4 15076.4,30244.6 15077)         |
|   7 | LINESTRING(30200.6 15059.4,30185.6 15058.6,30186 15048.8,30201.2 15049.4)   |
|  10 | LINESTRING(30179.6 15017.8,30181 15002.8,30190.8 15003.6,30189.6 15019)     |
|  11 | LINESTRING(30154.2 15000.4,30168.6 15004.8,30166 15014.2,30151.2 15009.8)   |
|  13 | LINESTRING(30105 15065.8,30108.4 15050.8,30118 15053,30114.6 15067.8)       |
|  21 | LINESTRING(30350.4 15828.8,30350.6 15845,30333.8 15845,30333.8 15828.8)     |
|  22 | LINESTRING(30350.6 15871.4,30350.6 15887.8,30334 15887.8,30334 15871.4)     |
|  23 | LINESTRING(30350.6 15914.2,30350.6 15930.4,30334 15930.4,30334 15914.2)     |
|  24 | LINESTRING(30290.2 15823,30290.2 15839.4,30273.4 15839.4,30273.4 15823)     |
|  25 | LINESTRING(30291.4 15866.2,30291.6 15882.4,30274.8 15882.4,30274.8 15866.2) |
|  26 | LINESTRING(30291.6 15918.2,30291.6 15934.4,30275 15934.4,30275 15918.2)     |
| 154 | LINESTRING(30276.2 15143.8,30261.4 15141,30263 15131.4,30278 15134)         |
| 155 | LINESTRING(30269.8 15084,30269.4 15093.4,30258.6 15093,30259 15083.4)       |
| 157 | LINESTRING(30128.2 15011,30113.2 15010.2,30113.6 15000.4,30128.8 15001)     |
| 249 | LINESTRING(30337.8 15938.6,30337.8 15946.8,30320.4 15946.8,30320.4 15938.4) |
+-----+-----------------------------------------------------------------------+
20 rows in set (0.46 sec)
```

When the index is not used, the execution time for this query rises from 0.00 seconds to 0.46 seconds.

In future releases, spatial indexes may also be used for optimizing other functions. See Section 7.5.4, "Functions for Testing Spatial Relations Between Geometric Objects."

## 7.7 MySQL Conformance and Compatibility

### 7.7.1 GIS Features That Are Not Yet Implemented

- Additional Metadata Views

  OpenGIS specifications propose several additional metadata views. For example, a system view named `GEOMETRY_COLUMNS` contains a description of geometry columns, one row for each geometry column in the database.

- The OpenGIS function `Length()` on `LineString` and `MultiLineString` currently should be called in MySQL as `GLength()`.

  The problem is that there is an existing SQL function `Length()` which calculates the length of string values, and sometimes it is not possible to distinguish whether the function is called in a textual or spatial context. We need either to solve this somehow, or decide on another function name.

# 8

# Stored Procedures and Functions

Stored procedures and functions are a new feature in MySQL version 5.0. A stored procedure is a set of SQL statements that can be stored in the server. Once this has been done, clients don't need to keep reissuing the individual statements but can refer to the stored procedure instead.

Some situations where stored procedures can be particularly useful:

- When multiple client applications are written in different languages or work on different platforms, but need to perform the same database operations.
- When security is paramount. Banks, for example, use stored procedures for all common operations. This provides a consistent and secure environment, and procedures can ensure that each operation is properly logged. In such a setup, applications and users would not get any access to the database tables directly, but can only execute specific stored procedures.

Stored procedures can provide improved performance because less information needs to be sent between the server and the client. The tradeoff is that this does increase the load on the database server system because more of the work is done on the server side and less is done on the client (application) side. Consider this if many client machines (such as Web servers) are serviced by only one or a few database servers.

Stored procedures also allow you to have libraries of functions in the database server. This is a feature shared by modern application languages that allow such design internally with, for example, classes. Using these client application language features is beneficial for the programmer even outside the scope of database use.

MySQL follows the SQL:2003 syntax for stored procedures, which is also used by IBM's DB2.

The MySQL implementation of stored procedures is still in progress. All syntax described in this chapter is supported and any limitations and extensions are documented where appropriate.

Stored procedures require the `proc` table in the `mysql` database. This table is created during the MySQL 5.0 installation procedure. If you are upgrading to MySQL 5.0 from an earlier version, be sure to update your grant tables to make sure the `proc` table exists.

## 8.1 Stored Procedure Syntax

Stored procedures and functions are routines that are created with `CREATE PROCEDURE` and `CREATE FUNCTION` statements. A routine is either a procedure or a function. A procedure is invoked using a `CALL` statement, and can only pass back values using output variables. Functions may return a scalar value and can be called from inside a statement just like any other function (that is, by invoking the function's name). Stored routines may call other stored routines.

At present, MySQL preserves context only for the default database. That is, if you say `USE` *`db_name`* within a procedure, the original default database is restored upon routine exit. A routine inherits the default database from the caller, so generally routines should either issue a `USE` *`db_name`* statement, or specify all tables with an explicit database reference; for example, *`db_name.tbl_name`*.

MySQL supports the very useful extension that allows the use of regular `SELECT` statements (that is, without using cursors or local variables) inside a stored procedure. The result set of such a query is simply sent directly to the client. Multiple `SELECT` statements generate multiple result sets, so the client must use a MySQL client library that supports multiple result sets. This means the client must use a client library from a version of MySQL at least as recent as 4.1.

The following section describes the syntax used to create, alter, drop, and query stored procedures and functions.

### 8.1.1 Maintaining Stored Procedures

#### 8.1.1.1 CREATE PROCEDURE and CREATE FUNCTION

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

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

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

type:
    Any valid MySQL data type

characteristic:
    LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | SQL SECURITY {DEFINER | INVOKER}
  | COMMENT 'string'

routine_body:
    Valid SQL procedure statement(s)
```

The `RETURNS` clause may be specified only for a `FUNCTION`. It is used to indicate the return type of the function, and the function body must contain a `RETURN value` statement.

The parameter list enclosed within parentheses must always be present. If there are no parameters, an empty parameter list of `()` should be used. Each parameter is an `IN` parameter by default. To specify otherwise for a parameter, use the keyword `OUT` or `INOUT` before the parameter name. Specifying `IN`, `OUT`, or `INOUT` is only valid for a `PROCEDURE`.

The `CREATE FUNCTION` statement is used in earlier versions of MySQL to support UDFs (User Defined Functions). UDFs continue to be supported, even with the existence of stored functions. A UDF can be regarded as an external stored function. However, do note that stored functions share their namespace with UDFs.

A framework for external stored procedures will be introduced in the near future. This will allow you to write stored procedures in languages other than SQL. Most likely, one of the first languages to be supported will be PHP because the core PHP engine is small, thread-safe, and can easily be embedded. Because the framework will be public, it is expected that many other languages will also be supported.

A function is considered "deterministic" if it always returns the same result for the same input parameters, and "not deterministic" otherwise. Currently, the `DETERMINISTIC` characteristic is accepted, but not yet used by the optimizer.

The `SQL SECURITY` characteristic can be used to specify whether the routine should be executed using the permissions of the user who creates the routine or the user who invokes it. The default value is `DEFINER`. This feature is new in SQL:2003.

MySQL does not yet use the `GRANT EXECUTE` privilege.

MySQL stores the `sql_mode` system variable setting that is in effect at the time a routine is created, and will always execute the routine with this setting in force.

The `COMMENT` clause is a MySQL extension, and may be used to describe the stored procedure. This information is displayed by the `SHOW CREATE PROCEDURE` and `SHOW CREATE FUNCTION` statements.

MySQL allows routines to contain DDL statements (such as `CREATE` and `DROP`) and SQL transaction statements (such as `COMMIT`). This is not required by the standard and is therefore implementation-specific.

**Note:** Currently, stored functions created with `CREATE FUNCTION` may not contain references to tables. Please note that this includes some `SET` statements, but excludes some `SELECT` statements. This limitation will be lifted as soon as possible.

The following is an example of a simple stored procedure that uses an `OUT` parameter. The example uses the `mysql` client `delimiter` command to change the statement delimiter from `;` to `//` while the procedure is being defined. This allows the `;` delimiter used in the procedure body to be passed through to the server rather than being interpreted by `mysql` itself.

```
mysql> delimiter //

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

mysql> delimiter ;

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

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

The following is an example of a function that takes a parameter, performs an operation using an SQL function, and returns the result:

```
mysql> delimiter //

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

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

#### 8.1.1.2 `ALTER PROCEDURE` and `ALTER FUNCTION`

```
ALTER {PROCEDURE | FUNCTION} sp_name [characteristic ...]

characteristic:
    NAME new_name
  | SQL SECURITY {DEFINER | INVOKER}
  | COMMENT 'string'
```

This statement can be used to rename a stored procedure or function, and to change its characteristics. More than one change may be specified in an `ALTER PROCEDURE` or `ALTER FUNCTION` statement.

#### 8.1.1.3 `DROP PROCEDURE` and `DROP FUNCTION`

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

This statement is used to drop a stored procedure or function. That is, the specified routine is removed from the server.

The `IF EXISTS` clause is a MySQL extension. It prevents an error from occurring if the procedure or function does not exist. A warning is produced that can be viewed with `SHOW WARNINGS`.

#### 8.1.1.4 `SHOW CREATE PROCEDURE` and `SHOW CREATE FUNCTION`

```
SHOW CREATE {PROCEDURE | FUNCTION} sp_name
```

This statement is a MySQL extension. Similar to `SHOW CREATE TABLE`, it returns the exact string that can be used to re-create the named routine.

### 8.1.2 `SHOW PROCEDURE STATUS` and `SHOW FUNCTION STATUS`

```
SHOW {PROCEDURE | FUNCTION} STATUS [LIKE 'pattern']
```

This statement is a MySQL extension. It returns characteristics of routines, such as the name, type, creator, and creation and modification dates. If no pattern is specified, the information for all stored procedures or all stored functions is listed, depending on which statement you use.

## 8.1.3 CALL

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

The `CALL` statement is used to invoke a procedure that was defined previously with `CREATE PROCEDURE`.

## 8.1.4 BEGIN ... END Compound Statement

```
[begin_label:] BEGIN
    statement(s)
END [end_label]
```

Stored routines may contain multiple statements, using a `BEGIN ... END` compound statement.

*`begin_label`* and *`end_label`* must be the same, if both are specified.

Please note that the optional `[NOT] ATOMIC` clause is not yet supported. This means that no transactional savepoint is set at the start of the instruction block and the `BEGIN` clause used in this context has no effect on the current transaction.

Using multiple statements requires that a client is able to send query strings containing the `;` statement delimiter. This is handled in the `mysql` command-line client with the `delimiter` command. Changing the `;` end-of-query delimiter (for example, to `//`) allows `;` to be used in a routine body.

## 8.1.5 DECLARE Statement

The `DECLARE` statement is used to define various items local to a routine: local variables (see Section 8.1.6, "Variables in Stored Procedures"), conditions and handlers (see Section 8.1.7, "Conditions and Handlers"), and cursors (see Section 8.1.8, "Cursors"). `SIGNAL` and `RESIGNAL` statements are not currently supported.

`DECLARE` may be used only inside a `BEGIN ... END` compound statement and must be at its start, before any other statements.

## 8.1.6 Variables in Stored Procedures

You may declare and use variables within a routine.

### 8.1.6.1 DECLARE Local Variables

```
DECLARE var_name[,...] type [DEFAULT value]
```

This statement is used to declare local variables. The scope of a variable is within the `BEGIN ... END` block.

### 8.1.6.2 Variable SET Statement

```
SET var_name = expr [,var_name = expr] ...]
```

The SET statement in stored procedures is an extended version of the general SET statement. Referenced variables may be ones declared inside a routine, or global server variables.

The SET statement in stored procedures is implemented as part of the pre-existing SET syntax. This allows an extended syntax of `SET a=x, b=y, ...` where different variable types (locally declared variables, server variables, and global and session server variables) can be mixed. This also allows combinations of local variables and some options that make sense only for global/system variables; in that case, the options are accepted but ignored.

### 8.1.6.3 SELECT ... INTO Statement

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

This SELECT syntax stores selected columns directly into variables. Therefore, only a single row may be retrieved. This statement is also extremely useful when used in combination with cursors.

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

## 8.1.7 Conditions and Handlers

Certain conditions may require specific handling. These conditions can relate to errors, as well as general flow control inside a routine.

### 8.1.7.1 DECLARE Conditions

```
DECLARE condition_name CONDITION FOR condition_value

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

This statement specifies conditions that will need specific handling. It associates a name with a specified error condition. The name can subsequently be used in a DECLARE HANDLER statement. See Section 8.1.7.2, "DECLARE Handlers."

In addition to SQLSTATE values, MySQL error codes are also supported.

### 8.1.7.2 DECLARE Handlers

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

handler_type:
    CONTINUE
  | EXIT
  | UNDO

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

This statement specifies handlers that each may deal with one or more conditions. If one of these conditions occurs, the specified statement is executed.

For a `CONTINUE` handler, execution of the current routine continues after execution of the handler statement. For an `EXIT` handler, execution of the current `BEGIN...END` compound statement is terminated. The `UNDO` handler type is not yet supported.

- `SQLWARNING` is shorthand for all SQLSTATE codes that begin with `01`.
- `NOT FOUND` is shorthand for all SQLSTATE codes that begin with `02`.
- `SQLEXCEPTION` is shorthand for all SQLSTATE codes not caught by `SQLWARNING` or `NOT FOUND`.

In addition to SQLSTATE values, MySQL error codes are also supported.

For example:

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

mysql> delimiter //

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

mysql> delimiter ;

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

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

Notice that `@x` is `3`, which shows that MySQL executed to the end of the procedure. If the line `DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;` had not been present, MySQL would have taken the default (`EXIT`) path after the second `INSERT` failed due to the `PRIMARY KEY` constraint, and `SELECT @x` would have returned `2`.

## 8.1.8 Cursors

Simple cursors are supported inside stored procedures and functions. The syntax is as in embedded SQL. Cursors are currently asensitive, read-only, and non-scrolling. Asensitive means that the server may or may not make a copy of its result table.

For example:

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

  OPEN cur1;
  OPEN cur2;

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

```
      ELSE
         INSERT INTO test.t3 VALUES (a,c);
      END IF;
    END IF;
  UNTIL done END REPEAT;

  CLOSE cur1;
  CLOSE cur2;
END
```

### 8.1.8.1 Declaring Cursors

DECLARE *cursor_name* CURSOR FOR *sql_statement*

Multiple cursors may be defined in a routine, but each must have a unique name.

### 8.1.8.2 Cursor OPEN Statement

OPEN *cursor_name*

This statement opens a previously declared cursor.

### 8.1.8.3 Cursor FETCH Statement

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

This statement fetches the next row (if a row exists) using the specified open cursor, and advances the cursor pointer.

### 8.1.8.4 Cursor CLOSE Statement

CLOSE *cursor_name*

This statement closes a previously opened cursor.

## 8.1.9 Flow Control Constructs

The IF, CASE, LOOP, WHILE, ITERATE, and LEAVE constructs are fully implemented.

These constructs may each contain either a single statement, or a block of statements using the BEGIN ... END compound statement. Constructs may be nested.

FOR loops are not currently supported.

### 8.1.9.1 IF Statement

```
IF search_condition THEN statement(s)
    [ELSEIF search_condition THEN statement(s)]
    ...
    [ELSE statement(s)]
END IF
```

`IF` implements a basic conditional construct. If the *`search_condition`* evaluates to true, the corresponding SQL statement is executed. If no *`search_condition`* matches, the statement in the `ELSE` clause is executed.

Please note that there is also an `IF()` function. See Section 5.2, "Control Flow Functions."

#### 8.1.9.2 CASE Statement

```
CASE case_value
    WHEN when_value THEN statement
    [WHEN when_value THEN statement ...]
    [ELSE statement]
END CASE
```

Or:

```
CASE
    WHEN search_condition THEN statement
    [WHEN search_condition THEN statement ...]
    [ELSE statement]
END CASE
```

`CASE` implements a complex conditional construct. If a *`search_condition`* evaluates to true, the corresponding SQL statement is executed. If no search condition matches, the statement in the `ELSE` clause is executed.

**Note:** The syntax of a `CASE` statement inside a stored procedure differs slightly from that of the SQL `CASE` expression. The `CASE` statement cannot have an `ELSE NULL` clause, and it is terminated with `END CASE` instead of `END`. See Section 5.2, "Control Flow Functions."

#### 8.1.9.3 LOOP Statement

```
[begin_label:] LOOP
    statement(s)
END LOOP [end_label]
```

`LOOP` implements a simple loop construct, enabling repeated execution of a particular statement or group of statements. The statements within the loop are repeated until the loop is exited; usually this is accomplished with a `LEAVE` statement.

*`begin_label`* and *`end_label`* must be the same, if both are specified.

#### 8.1.9.4 LEAVE Statement

```
LEAVE label
```

This statement is used to exit any flow control construct.

### 8.1.9.5 ITERATE Statement

```
ITERATE label
```

ITERATE can only appear within LOOP, REPEAT, and WHILE statements. ITERATE means "do the loop iteration again."

For example:

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

### 8.1.9.6 REPEAT Statement

```
[begin_label:] REPEAT
    statement(s)
UNTIL search_condition
END REPEAT [end_label]
```

The statements within a REPEAT statement are repeated until the *search_condition* is true.

*begin_label* and *end_label* must be the same, if both are specified.

For example:

```
mysql> delimiter //

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

mysql> delimiter ;

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

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

#### 8.1.9.7 `WHILE` Statement

```
[begin_label:] WHILE search_condition DO
    statement(s)
END WHILE [end_label]
```

The statements within a `WHILE` statement are repeated as long as the *`search_condition`* is true.

*`begin_label`* and *`end_label`* must be the same, if both are specified.

For example:

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

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

9

# Error Handling in MySQL

This chapter lists the errors that MySQL can return.

## 9.1 Error Returns

The following are error codes that may appear when you call MySQL from any host language.

The Name and Error Code columns correspond to definitions in the include/mysqld_error.h MySQL source file.

The SQLSTATE column corresponds to definitions in the include/sql_state.h MySQL source file.

The SQLSTATE error code is displayed only if you use MySQL version 4.1 and up. SQLSTATE codes were added for compatibility with X/Open, ANSI, and ODBC behavior.

A suggested text for each error code can be found in section 9.2, "Error Messages."

Because updates are frequent, it is possible that these files will contain additional error codes not listed here.

| Name | Error Code | SQLSTATE |
|---|---|---|
| ER_HASHCHK | 1000 | HY000 |
| ER_NISAMCHK | 1001 | HY000 |
| ER_NO | 1002 | HY000 |
| ER_YES | 1003 | HY000 |
| ER_CANT_CREATE_FILE | 1004 | HY000 |
| ER_CANT_CREATE_TABLE | 1005 | HY000 |
| ER_CANT_CREATE_DB | 1006 | HY000 |
| ER_DB_CREATE_EXISTS | 1007 | HY000 |
| ER_DB_DROP_EXISTS | 1008 | HY000 |
| ER_DB_DROP_DELETE | 1009 | HY000 |
| ER_DB_DROP_RMDIR | 1010 | HY000 |
| ER_CANT_DELETE_FILE | 1011 | HY000 |
| ER_CANT_FIND_SYSTEM_REC | 1012 | HY000 |
| ER_CANT_GET_STAT | 1013 | HY000 |
| ER_CANT_GET_WD | 1014 | HY000 |

| Name | Error Code | SQLSTATE |
|---|---|---|
| ER_CANT_LOCK | 1015 | HY000 |
| ER_CANT_OPEN_FILE | 1016 | HY000 |
| ER_FILE_NOT_FOUND | 1017 | HY000 |
| ER_CANT_READ_DIR | 1018 | HY000 |
| ER_CANT_SET_WD | 1019 | HY000 |
| ER_CHECKREAD | 1020 | HY000 |
| ER_DISK_FULL | 1021 | HY000 |
| ER_DUP_KEY | 1022 | 23000 |
| ER_ERROR_ON_CLOSE | 1023 | HY000 |
| ER_ERROR_ON_READ | 1024 | HY000 |
| ER_ERROR_ON_RENAME | 1025 | HY000 |
| ER_ERROR_ON_WRITE | 1026 | HY000 |
| ER_FILE_USED | 1027 | HY000 |
| ER_FILSORT_ABORT | 1028 | HY000 |
| ER_FORM_NOT_FOUND | 1029 | HY000 |
| ER_GET_ERRNO | 1030 | HY000 |
| ER_ILLEGAL_HA | 1031 | HY000 |
| ER_KEY_NOT_FOUND | 1032 | HY000 |
| ER_NOT_FORM_FILE | 1033 | HY000 |
| ER_NOT_KEYFILE | 1034 | HY000 |
| ER_OLD_KEYFILE | 1035 | HY000 |
| ER_OPEN_AS_READONLY | 1036 | HY000 |
| ER_OUTOFMEMORY | 1037 | HY001 |
| ER_OUT_OF_SORTMEMORY | 1038 | HY001 |
| ER_UNEXPECTED_EOF | 1039 | HY000 |
| ER_CON_COUNT_ERROR | 1040 | 08004 |
| ER_OUT_OF_RESOURCES | 1041 | 08004 |
| ER_BAD_HOST_ERROR | 1042 | 08S01 |
| ER_HANDSHAKE_ERROR | 1043 | 08S01 |
| ER_DBACCESS_DENIED_ERROR | 1044 | 42000 |
| ER_ACCESS_DENIED_ERROR | 1045 | 42000 |
| ER_NO_DB_ERROR | 1046 | 42000 |
| ER_UNKNOWN_COM_ERROR | 1047 | 08S01 |
| ER_BAD_NULL_ERROR | 1048 | 23000 |
| ER_BAD_DB_ERROR | 1049 | 42000 |
| ER_TABLE_EXISTS_ERROR | 1050 | 42S01 |
| ER_BAD_TABLE_ERROR | 1051 | 42S02 |
| ER_NON_UNIQ_ERROR | 1052 | 23000 |
| ER_SERVER_SHUTDOWN | 1053 | 08S01 |
| ER_BAD_FIELD_ERROR | 1054 | 42S22 |
| ER_WRONG_FIELD_WITH_GROUP | 1055 | 42000 |
| ER_WRONG_GROUP_FIELD | 1056 | 42000 |
| ER_WRONG_SUM_SELECT | 1057 | 42000 |
| ER_WRONG_VALUE_COUNT | 1058 | 21S01 |
| ER_TOO_LONG_IDENT | 1059 | 42000 |
| ER_DUP_FIELDNAME | 1060 | 42S21 |

| Name | Error Code | SQLSTATE |
|---|---|---|
| ER_DUP_KEYNAME | 1061 | 42000 |
| ER_DUP_ENTRY | 1062 | 23000 |
| ER_WRONG_FIELD_SPEC | 1063 | 42000 |
| ER_PARSE_ERROR | 1064 | 42000 |
| ER_EMPTY_QUERY | 1065 | 42000 |
| ER_NONUNIQ_TABLE | 1066 | 42000 |
| ER_INVALID_DEFAULT | 1067 | 42000 |
| ER_MULTIPLE_PRI_KEY | 1068 | 42000 |
| ER_TOO_MANY_KEYS | 1069 | 42000 |
| ER_TOO_MANY_KEY_PARTS | 1070 | 42000 |
| ER_TOO_LONG_KEY | 1071 | 42000 |
| ER_KEY_COLUMN_DOES_NOT_EXIST | 1072 | 42000 |
| ER_BLOB_USED_AS_KEY | 1073 | 42000 |
| ER_TOO_BIG_FIELDLENGTH | 1074 | 42000 |
| ER_WRONG_AUTO_KEY | 1075 | 42000 |
| ER_READY | 1076 | 00000 |
| ER_NORMAL_SHUTDOWN | 1077 | 00000 |
| ER_GOT_SIGNAL | 1078 | 00000 |
| ER_SHUTDOWN_COMPLETE | 1079 | 00000 |
| ER_FORCING_CLOSE | 1080 | 08S01 |
| ER_IPSOCK_ERROR | 1081 | 08S01 |
| ER_NO_SUCH_INDEX | 1082 | 42S12 |
| ER_WRONG_FIELD_TERMINATORS | 1083 | 42000 |
| ER_BLOBS_AND_NO_TERMINATED | 1084 | 42000 |
| ER_TEXTFILE_NOT_READABLE | 1085 | HY000 |
| ER_FILE_EXISTS_ERROR | 1086 | HY000 |
| ER_LOAD_INFO | 1087 | HY000 |
| ER_ALTER_INFO | 1088 | HY000 |
| ER_WRONG_SUB_KEY | 1089 | HY000 |
| ER_CANT_REMOVE_ALL_FIELDS | 1090 | 42000 |
| ER_CANT_DROP_FIELD_OR_KEY | 1091 | 42000 |
| ER_INSERT_INFO | 1092 | HY000 |
| ER_UPDATE_TABLE_USED | 1093 | HY000 |
| ER_NO_SUCH_THREAD | 1094 | HY000 |
| ER_KILL_DENIED_ERROR | 1095 | HY000 |
| ER_NO_TABLES_USED | 1096 | HY000 |
| ER_TOO_BIG_SET | 1097 | HY000 |
| ER_NO_UNIQUE_LOGFILE | 1098 | HY000 |
| ER_TABLE_NOT_LOCKED_FOR_WRITE | 1099 | HY000 |
| ER_TABLE_NOT_LOCKED | 1100 | HY000 |
| ER_BLOB_CANT_HAVE_DEFAULT | 1101 | 42000 |
| ER_WRONG_DB_NAME | 1102 | 42000 |
| ER_WRONG_TABLE_NAME | 1103 | 42000 |
| ER_TOO_BIG_SELECT | 1104 | 42000 |
| ER_UNKNOWN_ERROR | 1105 | HY000 |
| ER_UNKNOWN_PROCEDURE | 1106 | 42000 |

| Name | Error Code | SQLSTATE |
|---|---|---|
| ER_WRONG_PARAMCOUNT_TO_PROCEDURE | 1107 | 42000 |
| ER_WRONG_PARAMETERS_TO_PROCEDURE | 1108 | HY000 |
| ER_UNKNOWN_TABLE | 1109 | 42S02 |
| ER_FIELD_SPECIFIED_TWICE | 1110 | 42000 |
| ER_INVALID_GROUP_FUNC_USE | 1111 | 42000 |
| ER_UNSUPPORTED_EXTENSION | 1112 | 42000 |
| ER_TABLE_MUST_HAVE_COLUMNS | 1113 | 42000 |
| ER_RECORD_FILE_FULL | 1114 | HY000 |
| ER_UNKNOWN_CHARACTER_SET | 1115 | 42000 |
| ER_TOO_MANY_TABLES | 1116 | HY000 |
| ER_TOO_MANY_FIELDS | 1117 | HY000 |
| ER_TOO_BIG_ROWSIZE | 1118 | 42000 |
| ER_STACK_OVERRUN | 1119 | HY000 |
| ER_WRONG_OUTER_JOIN | 1120 | 42000 |
| ER_NULL_COLUMN_IN_INDEX | 1121 | 42000 |
| ER_CANT_FIND_UDF | 1122 | HY000 |
| ER_CANT_INITIALIZE_UDF | 1123 | HY000 |
| ER_UDF_NO_PATHS | 1124 | HY000 |
| ER_UDF_EXISTS | 1125 | HY000 |
| ER_CANT_OPEN_LIBRARY | 1126 | HY000 |
| ER_CANT_FIND_DL_ENTRY | 1127 | HY000 |
| ER_FUNCTION_NOT_DEFINED | 1128 | HY000 |
| ER_HOST_IS_BLOCKED | 1129 | HY000 |
| ER_HOST_NOT_PRIVILEGED | 1130 | HY000 |
| ER_PASSWORD_ANONYMOUS_USER | 1131 | 42000 |
| ER_PASSWORD_NOT_ALLOWED | 1132 | 42000 |
| ER_PASSWORD_NO_MATCH | 1133 | 42000 |
| ER_UPDATE_INFO | 1134 | HY000 |
| ER_CANT_CREATE_THREAD | 1135 | HY000 |
| ER_WRONG_VALUE_COUNT_ON_ROW | 1136 | 21S01 |
| ER_CANT_REOPEN_TABLE | 1137 | HY000 |
| ER_INVALID_USE_OF_NULL | 1138 | 42000 |
| ER_REGEXP_ERROR | 1139 | 42000 |
| ER_MIX_OF_GROUP_FUNC_AND_FIELDS | 1140 | 42000 |
| ER_NONEXISTING_GRANT | 1141 | 42000 |
| ER_TABLEACCESS_DENIED_ERROR | 1142 | 42000 |
| ER_COLUMNACCESS_DENIED_ERROR | 1143 | 42000 |
| ER_ILLEGAL_GRANT_FOR_TABLE | 1144 | 42000 |
| ER_GRANT_WRONG_HOST_OR_USER | 1145 | 42000 |
| ER_NO_SUCH_TABLE | 1146 | 42S02 |
| ER_NONEXISTING_TABLE_GRANT | 1147 | 42000 |
| ER_NOT_ALLOWED_COMMAND | 1148 | 42000 |
| ER_SYNTAX_ERROR | 1149 | 42000 |
| ER_DELAYED_CANT_CHANGE_LOCK | 1150 | HY000 |
| ER_TOO_MANY_DELAYED_THREADS | 1151 | HY000 |
| ER_ABORTING_CONNECTION | 1152 | 08S01 |

| Name | Error Code | SQLSTATE |
|---|---|---|
| ER_NET_PACKET_TOO_LARGE | 1153 | 08S01 |
| ER_NET_READ_ERROR_FROM_PIPE | 1154 | 08S01 |
| ER_NET_FCNTL_ERROR | 1155 | 08S01 |
| ER_NET_PACKETS_OUT_OF_ORDER | 1156 | 08S01 |
| ER_NET_UNCOMPRESS_ERROR | 1157 | 08S01 |
| ER_NET_READ_ERROR | 1158 | 08S01 |
| ER_NET_READ_INTERRUPTED | 1159 | 08S01 |
| ER_NET_ERROR_ON_WRITE | 1160 | 08S01 |
| ER_NET_WRITE_INTERRUPTED | 1161 | 08S01 |
| ER_TOO_LONG_STRING | 1162 | 42000 |
| ER_TABLE_CANT_HANDLE_BLOB | 1163 | 42000 |
| ER_TABLE_CANT_HANDLE_AUTO_INCREMENT | 1164 | 42000 |
| ER_DELAYED_INSERT_TABLE_LOCKED | 1165 | HY000 |
| ER_WRONG_COLUMN_NAME | 1166 | 42000 |
| ER_WRONG_KEY_COLUMN | 1167 | 42000 |
| ER_WRONG_MRG_TABLE | 1168 | HY000 |
| ER_DUP_UNIQUE | 1169 | 23000 |
| ER_BLOB_KEY_WITHOUT_LENGTH | 1170 | 42000 |
| ER_PRIMARY_CANT_HAVE_NULL | 1171 | 42000 |
| ER_TOO_MANY_ROWS | 1172 | 42000 |
| ER_REQUIRES_PRIMARY_KEY | 1173 | 42000 |
| ER_NO_RAID_COMPILED | 1174 | HY000 |
| ER_UPDATE_WITHOUT_KEY_IN_SAFE_MODE | 1175 | HY000 |
| ER_KEY_DOES_NOT_EXITS | 1176 | HY000 |
| ER_CHECK_NO_SUCH_TABLE | 1177 | 42000 |
| ER_CHECK_NOT_IMPLEMENTED | 1178 | 42000 |
| ER_CANT_DO_THIS_DURING_AN_TRANSACTION | 1179 | 25000 |
| ER_ERROR_DURING_COMMIT | 1180 | HY000 |
| ER_ERROR_DURING_ROLLBACK | 1181 | HY000 |
| ER_ERROR_DURING_FLUSH_LOGS | 1182 | HY000 |
| ER_ERROR_DURING_CHECKPOINT | 1183 | HY000 |
| ER_NEW_ABORTING_CONNECTION | 1184 | 08S01 |
| ER_DUMP_NOT_IMPLEMENTED | 1185 | HY000 |
| ER_FLUSH_MASTER_BINLOG_CLOSED | 1186 | HY000 |
| ER_INDEX_REBUILD | 1187 | HY000 |
| ER_MASTER | 1188 | HY000 |
| ER_MASTER_NET_READ | 1189 | 08S01 |
| ER_MASTER_NET_WRITE | 1190 | 08S01 |
| ER_FT_MATCHING_KEY_NOT_FOUND | 1191 | HY000 |
| ER_LOCK_OR_ACTIVE_TRANSACTION | 1192 | HY000 |
| ER_UNKNOWN_SYSTEM_VARIABLE | 1193 | HY000 |
| ER_CRASHED_ON_USAGE | 1194 | HY000 |
| ER_CRASHED_ON_REPAIR | 1195 | HY000 |
| ER_WARNING_NOT_COMPLETE_ROLLBACK | 1196 | HY000 |
| ER_TRANS_CACHE_FULL | 1197 | HY000 |
| ER_SLAVE_MUST_STOP | 1198 | HY000 |

| Name | Error Code | SQLSTATE |
|---|---|---|
| ER_SLAVE_NOT_RUNNING | 1199 | HY000 |
| ER_BAD_SLAVE | 1200 | HY000 |
| ER_MASTER_INFO | 1201 | HY000 |
| ER_SLAVE_THREAD | 1202 | HY000 |
| ER_TOO_MANY_USER_CONNECTIONS | 1203 | 42000 |
| ER_SET_CONSTANTS_ONLY | 1204 | HY000 |
| ER_LOCK_WAIT_TIMEOUT | 1205 | HY000 |
| ER_LOCK_TABLE_FULL | 1206 | HY000 |
| ER_READ_ONLY_TRANSACTION | 1207 | 25000 |
| ER_DROP_DB_WITH_READ_LOCK | 1208 | HY000 |
| ER_CREATE_DB_WITH_READ_LOCK | 1209 | HY000 |
| ER_WRONG_ARGUMENTS | 1210 | HY000 |
| ER_NO_PERMISSION_TO_CREATE_USER | 1211 | 42000 |
| ER_UNION_TABLES_IN_DIFFERENT_DIR | 1212 | HY000 |
| ER_LOCK_DEADLOCK | 1213 | 40001 |
| ER_TABLE_CANT_HANDLE_FULLTEXT | 1214 | HY000 |
| ER_CANNOT_ADD_FOREIGN | 1215 | HY000 |
| ER_NO_REFERENCED_ROW | 1216 | 23000 |
| ER_ROW_IS_REFERENCED | 1217 | 23000 |
| ER_CONNECT_TO_MASTER | 1218 | 08S01 |
| ER_QUERY_ON_MASTER | 1219 | HY000 |
| ER_ERROR_WHEN_EXECUTING_COMMAND | 1220 | HY000 |
| ER_WRONG_USAGE | 1221 | HY000 |
| ER_WRONG_NUMBER_OF_COLUMNS_IN_SELECT | 1222 | 21000 |
| ER_CANT_UPDATE_WITH_READLOCK | 1223 | HY000 |
| ER_MIXING_NOT_ALLOWED | 1224 | HY000 |
| ER_DUP_ARGUMENT | 1225 | HY000 |
| ER_USER_LIMIT_REACHED | 1226 | 42000 |
| ER_SPECIFIC_ACCESS_DENIED_ERROR | 1227 | HY000 |
| ER_LOCAL_VARIABLE | 1228 | HY000 |
| ER_GLOBAL_VARIABLE | 1229 | HY000 |
| ER_NO_DEFAULT | 1230 | 42000 |
| ER_WRONG_VALUE_FOR_VAR | 1231 | 42000 |
| ER_WRONG_TYPE_FOR_VAR | 1232 | 42000 |
| ER_VAR_CANT_BE_READ | 1233 | HY000 |
| ER_CANT_USE_OPTION_HERE | 1234 | 42000 |
| ER_NOT_SUPPORTED_YET | 1235 | 42000 |
| ER_MASTER_FATAL_ERROR_READING_BINLOG | 1236 | HY000 |
| ER_SLAVE_IGNORED_TABLE | 1237 | HY000 |
| ER_INCORRECT_GLOBAL_LOCAL_VAR | 1238 | HY000 |
| ER_WRONG_FK_DEF | 1239 | 42000 |
| ER_KEY_REF_DO_NOT_MATCH_TABLE_REF | 1240 | HY000 |
| ER_OPERAND_COLUMNS | 1241 | 21000 |
| ER_SUBQUERY_NO_1_ROW | 1242 | 21000 |
| ER_UNKNOWN_STMT_HANDLER | 1243 | HY000 |
| ER_CORRUPT_HELP_DB | 1244 | HY000 |

| Name | Error Code | SQLSTATE |
|---|---|---|
| ER_CYCLIC_REFERENCE | 1245 | HY000 |
| ER_AUTO_CONVERT | 1246 | HY000 |
| ER_ILLEGAL_REFERENCE | 1247 | 42S22 |
| ER_DERIVED_MUST_HAVE_ALIAS | 1248 | 42000 |
| ER_SELECT_REDUCED | 1249 | 01000 |
| ER_TABLENAME_NOT_ALLOWED_HERE | 1250 | 42000 |
| ER_NOT_SUPPORTED_AUTH_MODE | 1251 | 08004 |
| ER_SPATIAL_CANT_HAVE_NULL | 1252 | 42000 |
| ER_COLLATION_CHARSET_MISMATCH | 1253 | 42000 |
| ER_SLAVE_WAS_RUNNING | 1254 | HY000 |
| ER_SLAVE_WAS_NOT_RUNNING | 1255 | HY000 |
| ER_TOO_BIG_FOR_UNCOMPRESS | 1256 | HY000 |
| ER_ZLIB_Z_MEM_ERROR | 1257 | HY000 |
| ER_ZLIB_Z_BUF_ERROR | 1258 | HY000 |
| ER_ZLIB_Z_DATA_ERROR | 1259 | HY000 |
| ER_CUT_VALUE_GROUP_CONCAT | 1260 | HY000 |
| ER_WARN_TOO_FEW_RECORDS | 1261 | 01000 |
| ER_WARN_TOO_MANY_RECORDS | 1262 | 01000 |
| ER_WARN_NULL_TO_NOTNULL | 1263 | 01000 |
| ER_WARN_DATA_OUT_OF_RANGE | 1264 | 01000 |
| ER_WARN_DATA_TRUNCATED | 1265 | 01000 |
| ER_WARN_USING_OTHER_HANDLER | 1266 | HY000 |
| ER_CANT_AGGREGATE_2COLLATIONS | 1267 | HY000 |
| ER_DROP_USER | 1268 | HY000 |
| ER_REVOKE_GRANTS | 1269 | HY000 |
| ER_CANT_AGGREGATE_3COLLATIONS | 1270 | HY000 |
| ER_CANT_AGGREGATE_NCOLLATIONS | 1271 | HY000 |
| ER_VARIABLE_IS_NOT_STRUCT | 1272 | HY000 |
| ER_UNKNOWN_COLLATION | 1273 | HY000 |
| ER_SLAVE_IGNORED_SSL_PARAMS | 1274 | HY000 |
| ER_SERVER_IS_IN_SECURE_AUTH_MODE | 1275 | HY000 |
| ER_WARN_FIELD_RESOLVED | 1276 | HY000 |
| ER_BAD_SLAVE_UNTIL_COND | 1277 | HY000 |
| ER_MISSING_SKIP_SLAVE | 1278 | HY000 |
| ER_UNTIL_COND_IGNORED | 1279 | HY000 |
| ER_WRONG_NAME_FOR_INDEX | 1280 | 42000 |
| ER_WRONG_NAME_FOR_CATALOG | 1281 | 42000 |
| ER_WARN_QC_RESIZE | 1282 | HY000 |
| ER_BAD_FT_COLUMN | 1283 | HY000 |
| ER_UNKNOWN_KEY_CACHE | 1284 | HY000 |
| ER_WARN_HOSTNAME_WONT_WORK | 1285 | HY000 |
| ER_UNKNOWN_STORAGE_ENGINE | 1286 | 42000 |
| ER_WARN_DEPRECATED_SYNTAX | 1287 | HY000 |
| ER_NON_UPDATABLE_TABLE | 1288 | HY000 |
| ER_FEATURE_DISABLED | 1289 | HY000 |
| ER_OPTION_PREVENTS_STATEMENT | 1290 | HY000 |

| Name | Error Code | SQLSTATE |
|---|---|---|
| ER_DUPLICATED_VALUE_IN_TYPE | 1291 | HY000 |
| ER_TRUNCATED_WRONG_VALUE | 1292 | HY000 |
| ER_TOO_MUCH_AUTO_TIMESTAMP_COLS | 1293 | HY000 |
| ER_INVALID_ON_UPDATE | 1294 | HY000 |
| ER_UNSUPPORTED_PS | 1295 | HY000 |
| ER_SP_NO_RECURSIVE_CREATE | 1296 | 2F003 |
| ER_SP_ALREADY_EXISTS | 1297 | 42000 |
| ER_SP_DOES_NOT_EXIST | 1298 | 42000 |
| ER_SP_DROP_FAILED | 1299 | HY000 |
| ER_SP_STORE_FAILED | 1300 | HY000 |
| ER_SP_LILABEL_MISMATCH | 1301 | 42000 |
| ER_SP_LABEL_REDEFINE | 1302 | 42000 |
| ER_SP_LABEL_MISMATCH | 1303 | 42000 |
| ER_SP_UNINIT_VAR | 1304 | 01000 |
| ER_SP_BADSELECT | 1305 | 0A000 |
| ER_SP_BADRETURN | 1306 | 42000 |
| ER_SP_BADSTATEMENT | 1307 | 0A000 |
| ER_UPDATE_LOG_DEPRECATED_IGNORED | 1308 | 42000 |
| ER_UPDATE_LOG_DEPRECATED_TRANSLATED | 1309 | 42000 |
| ER_QUERY_INTERRUPTED | 1310 | 70100 |
| ER_SP_WRONG_NO_OF_ARGS | 1311 | 42000 |
| ER_SP_COND_MISMATCH | 1312 | 42000 |
| ER_SP_NORETURN | 1313 | 42000 |
| ER_SP_NORETURNEND | 1314 | 2F005 |
| ER_SP_BAD_CURSOR_QUERY | 1315 | 42000 |
| ER_SP_BAD_CURSOR_SELECT | 1316 | 42000 |
| ER_SP_CURSOR_MISMATCH | 1317 | 42000 |
| ER_SP_CURSOR_ALREADY_OPEN | 1318 | 24000 |
| ER_SP_CURSOR_NOT_OPEN | 1319 | 24000 |
| ER_SP_UNDECLARED_VAR | 1320 | 42000 |
| ER_SP_WRONG_NO_OF_FETCH_ARGS | 1321 | HY000 |
| ER_SP_FETCH_NO_DATA | 1322 | 02000 |
| ER_SP_DUP_PARAM | 1323 | 42000 |
| ER_SP_DUP_VAR | 1324 | 42000 |
| ER_SP_DUP_COND | 1325 | 42000 |
| ER_SP_DUP_CURS | 1326 | 42000 |
| ER_SP_CANT_ALTER | 1327 | HY000 |
| ER_SP_SUBSELECT_NYI | 1328 | 0A000 |
| ER_SP_NO_USE | 1329 | 42000 |
| ER_SP_VARCOND_AFTER_CURSHNDLR | 1330 | 42000 |
| ER_SP_CURSOR_AFTER_HANDLER | 1331 | 42000 |
| ER_SP_CASE_NOT_FOUND | 1332 | 20000 |
| ER_FPARSER_TOO_BIG_FILE | 1333 | HY000 |
| ER_FPARSER_BAD_HEADER | 1334 | HY000 |
| ER_FPARSER_EOF_IN_COMMENT | 1335 | HY000 |
| ER_FPARSER_ERROR_IN_PARAMETER | 1336 | HY000 |
| ER_FPARSER_EOF_IN_UNKNOWN_PARAMETER | 1337 | HY000 |

## 9.2 Error Messages

Following are error messages that may appear when you call MySQL from any host language. %d or %s represent numbers or strings that are substituted into the messages.

| Error Code | Error Message |
|---|---|
| 1000 | hashchk |
| 1001 | isamchk |
| 1002 | NO |
| 1003 | YES |
| 1004 | Can’t create file ‘%s’ (errno: %d) |
| 1005 | Can’t create table ‘%s’ (errno: %d) |
| 1006 | Can’t create database ‘%s’ (errno: %d) |
| 1007 | Can’t create database ‘%s’; database exists |
| 1008 | Can’t drop database ‘%s’; database doesn’t exist |
| 1009 | Error dropping database (can’t delete ‘%s’, errno: %d) |
| 1010 | Error dropping database (can’t rmdir ‘%s’, errno: %d) |
| 1011 | Error on delete of ‘%s’ (errno: %d) |
| 1012 | Can’t read record in system table |
| 1013 | Can’t get status of ‘%s’ (errno: %d) |
| 1014 | Can’t get working directory (errno: %d) |
| 1015 | Can’t lock file (errno: %d) |
| 1016 | Can’t open file: ‘%s’ (errno: %d) |
| 1017 | Can’t find file: ‘%s’ (errno: %d) |
| 1018 | Can’t read dir of ‘%s’ (errno: %d) |
| 1019 | Can’t change dir to ‘%s’ (errno: %d) |
| 1020 | Record has changed since last read in table ‘%s’ |
| 1021 | Disk full (%s); waiting for someone to free some space... |
| 1022 | Can’t write; duplicate key in table ‘%s’ |
| 1023 | Error on close of ‘%s’ (errno: %d) |
| 1024 | Error reading file ‘%s’ (errno: %d) |
| 1025 | Error on rename of ‘%s’ to ‘%s’ (errno: %d) |
| 1026 | Error writing file ‘%s’ (errno: %d) |
| 1027 | ‘%s’ is locked against change |
| 1028 | Sort aborted |
| 1029 | View ‘%s’ doesn’t exist for ‘%s’ |
| 1030 | Got error %d from storage engine |
| 1031 | Table storage engine for ‘%s’ doesn’t have this option |
| 1032 | Can’t find record in ‘%s’ |
| 1033 | Incorrect information in file: ‘%s’ |
| 1034 | Incorrect key file for table ‘%s’; try to repair it |
| 1035 | Old key file for table ‘%s’; repair it! |
| 1036 | Table ‘%s’ is read only |
| 1037 | Out of memory; restart server and try again (needed %d bytes) |
| 1038 | Out of sort memory; increase server sort buffer size |
| 1039 | Unexpected EOF found when reading file ‘%s’ (errno: %d) |
| 1040 | Too many connections |

| Error Code | Error Message |
|---|---|
| 1041 | `Out of memory; check if mysqld or some other process uses all available memory; if not, you may have to use ‘ulimit’ to allow mysqld to use more memory or you can add more swap space` |
| 1042 | `Can’t get hostname for your address` |
| 1043 | `Bad handshake` |
| 1044 | `Access denied for user ‘%s’@’%s’ to database ‘%s’` |
| 1045 | `Access denied for user ‘%s’@’%s’ (using password: %s)` |
| 1046 | `No database selected` |
| 1047 | `Unknown command` |
| 1048 | `Column ‘%s’ cannot be null` |
| 1049 | `Unknown database ‘%s’` |
| 1050 | `Table ‘%s’ already exists` |
| 1051 | `Unknown table ‘%s’` |
| 1052 | `Column ‘%s’ in %s is ambiguous` |
| 1053 | `Server shutdown in progress` |
| 1054 | `Unknown column ‘%s’ in ‘%s’` |
| 1055 | `‘%s’ isn’t in GROUP BY` |
| 1056 | `Can’t group on ‘%s’` |
| 1057 | `Statement has sum functions and columns in same statement` |
| 1058 | `Column count doesn’t match value count` |
| 1059 | `Identifier name ‘%s’ is too long` |
| 1060 | `Duplicate column name ‘%s’` |
| 1061 | `Duplicate key name ‘%s’` |
| 1062 | `Duplicate entry ‘%s’ for key %d` |
| 1063 | `Incorrect column specifier for column ‘%s’` |
| 1064 | `%s near ‘%s’ at line %d` |
| 1065 | `Query was empty` |
| 1066 | `Not unique table/alias: ‘%s’` |
| 1067 | `Invalid default value for ‘%s’` |
| 1068 | `Multiple primary key defined` |
| 1069 | `Too many keys specified; max %d keys allowed` |
| 1070 | `Too many key parts specified; max %d parts allowed` |
| 1071 | `Specified key was too long; max key length is %d bytes` |
| 1072 | `Key column ‘%s’ doesn’t exist in table` |
| 1073 | `BLOB column ‘%s’ can’t be used in key specification with the used table type` |
| 1074 | `Column length too big for column ‘%s’ (max = %d); use BLOB instead` |
| 1075 | `Incorrect table definition; there can be only one auto column and it must be defined as a key` |
| 1076 | `%s: ready for connections. Version: ‘%s’  socket: ‘%s’ port: %d` |
| 1077 | `%s: Normal shutdown` |
| 1078 | `%s: Got signal %d. Aborting!` |
| 1079 | `%s: Shutdown complete` |
| 1080 | `%s: Forcing close of thread %ld  user: ‘%s’` |

| Error Code | Error Message |
|---|---|
| 1081 | Can't create IP socket |
| 1082 | Table '%s' has no index like the one used in CREATE INDEX; :ecreate the table |
| 1083 | Field separator argument is not what is expected; check the manual |
| 1084 | You can't use fixed rowlength with BLOBs; please use 'fields terminated by' |
| 1085 | The file '%s' must be in the database directory or be readable by all |
| 1086 | File '%s' already exists |
| 1087 | Records: %ld  Deleted: %ld  Skipped: %ld  Warnings: %ld |
| 1088 | Records: %ld  Duplicates: %ld |
| 1089 | Incorrect sub part key; the used key part isn't a string, the used length is longer than the key part, or the storage engine doesn't support unique sub keys |
| 1090 | You can't delete all columns with ALTER TABLE; use DROP TABLE instead |
| 1091 | Can't DROP '%s'; check that column/key exists |
| 1092 | Records: %ld  Duplicates: %ld  Warnings: %ld |
| 1093 | You can't specify target table '%s' for update in FROM clause |
| 1094 | Unknown thread id: %lu |
| 1095 | You are not owner of thread %lu |
| 1096 | No tables used |
| 1097 | Too many strings for column %s and SET |
| 1098 | Can't generate a unique log-filename %s.(1-999) |
| 1099 | Table '%s' was locked with a READ lock and can't be updated |
| 1100 | Table '%s' was not locked with LOCK TABLES |
| 1101 | BLOB/TEXT column '%s' can't have a default value |
| 1102 | Incorrect database name '%s' |
| 1103 | Incorrect table name '%s' |
| 1104 | The SELECT would examine more than MAX_JOIN_SIZE rows; check your WHERE and use SET SQL_BIG_SELECTS=1 or SET SQL_MAX_JOIN_SIZE=# if the SELECT is okay |
| 1105 | Unknown error |
| 1106 | Unknown procedure '%s' |
| 1107 | Incorrect parameter count to procedure '%s' |
| 1108 | Incorrect parameters to procedure '%s' |
| 1109 | Unknown table '%s' in %s |
| 1110 | Column '%s' specified twice |
| 1111 | Invalid use of group function |
| 1112 | Table '%s' uses an extension that doesn't exist in this MySQL version |
| 1113 | A table must have at least 1 column |
| 1114 | The table '%s' is full |
| 1115 | Unknown character set: '%s' |
| 1116 | Too many tables; MySQL can only use %d tables in a join |
| 1117 | Too many columns |

| Error Code | Error Message |
|---|---|
| 1118 | Row size too large. The maximum row size for the used table type, not counting BLOBs, is %ld. You have to change some columns to TEXT or BLOBs |
| 1119 | Thread stack overrun:  Used: %ld of a %ld stack.  Use 'mysqld -O thread_stack=#' to specify a bigger stack if needed |
| 1120 | Cross dependency found in OUTER JOIN; examine your ON condi- tions |
| 1121 | Column '%s' is used with UNIQUE or INDEX but is not defined as NOT NULL |
| 1122 | Can't load function '%s' |
| 1123 | Can't initialize function '%s'; %s |
| 1124 | No paths allowed for shared library |
| 1125 | Function '%s' already exists |
| 1126 | Can't open shared library '%s' (errno: %d %s) |
| 1127 | Can't find function '%s' in library' |
| 1128 | Function '%s' is not defined |
| 1129 | Host '%s' is blocked because of many connection errors; unblock with 'mysqladmin flush-hosts' |
| 1130 | Host '%s' is not allowed to connect to this MySQL server |
| 1131 | You are using MySQL as an anonymous user and anonymous users are not allowed to change passwords |
| 1132 | You must have privileges to update tables in the mysql data- base to be able to change passwords for others |
| 1133 | Can't find any matching row in the user table |
| 1134 | Rows matched: %ld  Changed: %ld  Warnings: %ld |
| 1135 | Can't create a new thread (errno %d); if you are not out of available memory, you can consult the manual for a possible OS-dependent bug |
| 1136 | Column count doesn't match value count at row %ld |
| 1137 | Can't reopen table: '%s' |
| 1138 | Invalid use of NULL value |
| 1139 | Got error '%s' from regexp |
| 1140 | Mixing of GROUP columns (MIN(),MAX(),COUNT(),...) with no GROUP columns is illegal if there is no GROUP BY clause |
| 1141 | There is no such grant defined for user '%s' on host '%s' |
| 1142 | %s command denied to user '%s'@'%s' for table '%s' |
| 1143 | %s command denied to user '%s'@'%s' for column '%s' in table '%s' |
| 1144 | Illegal GRANT/REVOKE command; please consult the manual to see which privileges can be used |
| 1145 | The host or user argument to GRANT is too long |
| 1146 | Table '%s.%s' doesn't exist |
| 1147 | There is no such grant defined for user '%s' on host '%s' on table '%s' |
| 1148 | The used command is not allowed with this MySQL version |

| Error Code | Error Message |
|---|---|
| 1149 | You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use |
| 1150 | Delayed insert thread couldn't get requested lock for table %s |
| 1151 | Too many delayed threads in use |
| 1152 | Aborted connection %ld to db: '%s' user: '%s' (%s) |
| 1153 | Got a packet bigger than 'max_allowed_packet' bytes |
| 1154 | Got a read error from the connection pipe |
| 1155 | Got an error from fcntl() |
| 1156 | Got packets out of order |
| 1157 | Couldn't uncompress communication packet |
| 1158 | Got an error reading communication packets |
| 1159 | Got timeout reading communication packets |
| 1160 | Got an error writing communication packets |
| 1161 | Got timeout writing communication packets |
| 1162 | Result string is longer than 'max_allowed_packet' bytes |
| 1163 | The used table type doesn't support BLOB/TEXT columns |
| 1164 | The used table type doesn't support AUTO_INCREMENT columns |
| 1165 | INSERT DELAYED can't be used with table '%s' because it is locked with LOCK TABLES |
| 1166 | Incorrect column name '%s' |
| 1167 | The used storage engine can't index column '%s' |
| 1168 | All tables in the MERGE table are not identically defined |
| 1169 | Can't write, because of unique constraint, to table '%s' |
| 1170 | BLOB/TEXT column '%s' used in key specification without a key length |
| 1171 | All parts of a PRIMARY KEY must be NOT NULL; if you need NULL in a key, use UNIQUE instead |
| 1172 | Result consisted of more than one row |
| 1173 | This table type requires a primary key |
| 1174 | This version of MySQL is not compiled with RAID support |
| 1175 | You are using safe update mode and you tried to update a table without a WHERE that uses a KEY column |
| 1176 | Key '%s' doesn't exist in table '%s' |
| 1177 | Can't open table |
| 1178 | The storage engine for the table doesn't support %s |
| 1179 | You are not allowed to execute this command in a transaction |
| 1180 | Got error %d during COMMIT |
| 1181 | Got error %d during ROLLBACK |
| 1182 | Got error %d during FLUSH_LOGS |
| 1183 | Got error %d during CHECKPOINT |
| 1184 | Aborted connection %ld to db: '%s' user: '%s' host: `%s' (%s) |
| 1185 | The storage engine for the table does not support binary table dump |

| Error Code | Error Message |
|---|---|
| 1186 | Binlog closed, cannot RESET MASTER |
| 1187 | Failed rebuilding the index of  dumped table '%s' |
| 1188 | Error from master: '%s' |
| 1189 | Net error reading from master |
| 1190 | Net error writing to master |
| 1191 | Can't find FULLTEXT index matching the column list |
| 1192 | Can't execute the given command because you have active locked tables or an active transaction |
| 1193 | Unknown system variable '%s' |
| 1194 | Table '%s' is marked as crashed and should be repaired |
| 1195 | Table '%s' is marked as crashed and last (automatic?) repair failed |
| 1196 | Some non-transactional changed tables couldn't be rolled back |
| 1197 | Multi-statement transaction required more than 'max_binlog_cache_size' bytes of storage; increase this mysqld variable and try again |
| 1198 | This operation cannot be performed with a running slave; run STOP SLAVE first |
| 1199 | This operation requires a running slave; configure slave and do START SLAVE |
| 1200 | The server is not configured as slave; fix in config file or with CHANGE MASTER TO |
| 1201 | Could not initialize master info structure; more error messages can be found in the MySQL error log |
| 1202 | Could not create slave thread; check system resources |
| 1203 | User %s has already more than 'max_user_connections' active connections |
| 1204 | You may only use constant expressions with SET |
| 1205 | Lock wait timeout exceeded; try restarting transaction |
| 1206 | The total number of locks exceeds the lock table size |
| 1207 | Update locks cannot be acquired during a READ UNCOMMITTED transaction |
| 1208 lock | DROP DATABASE not allowed while thread is holding global read |
| 1209 | CREATE DATABASE not allowed while thread is holding global read lock |
| 1210 | Incorrect arguments to %s |
| 1211 | '%s'@'%s' is not allowed to create new users |
| 1212 | Incorrect table definition; all MERGE tables must be in the same database |
| 1213 | Deadlock found when trying to get lock; try restarting transaction |
| 1214 | The used table type doesn't support FULLTEXT indexes |
| 1215 | Cannot add foreign key constraint |
| 1216 | Cannot add or update a child row: a foreign key constraint fails |
| 1217 | Cannot delete or update a parent row: a foreign key constraint fails |

| Error Code | Error Message |
|---|---|
| 1218 | Error connecting to master: %s |
| 1219 | Error running query on master: %s |
| 1220 | Error when executing command %s: %s |
| 1221 | Incorrect usage of %s and %s |
| 1222 | The used SELECT statements have a different number of columns |
| 1223 | Can't execute the query because you have a conflicting read lock |
| 1224 | Mixing of transactional and non-transactional tables is disabled |
| 1225 | Option '%s' used twice in statement |
| 1226 | User '%s' has exceeded the '%s' resource (current value: %ld) |
| 1227 | Access denied; you need the %s privilege for this operation |
| 1228 | Variable '%s' is a SESSION variable and can't be used with SET GLOBAL |
| 1229 | Variable '%s' is a GLOBAL variable and should be set with SET GLOBAL |
| 1230 | Variable '%s' doesn't have a default value |
| 1231 | Variable '%s' can't be set to the value of '%s' |
| 1232 | Incorrect argument type to variable '%s' |
| 1233 | Variable '%s' can only be set, not read |
| 1234 | Incorrect usage/placement of '%s' |
| 1235 | This version of MySQL doesn't yet support '%s' |
| 1236 | Got fatal error %d: '%s' from master when reading data from binary log |
| 1237 | Slave SQL thread ignored the query because of replicate-*-table rules |
| 1238 | Variable '%s' is a %s variable |
| 1239 | Incorrect foreign key definition for '%s': %s |
| 1240 | Key reference and table reference don't match |
| 1241 | Operand should contain %d column(s) |
| 1242 | Subquery returns more than 1 row |
| 1243 | Unknown prepared statement handler (%.*s) given to %s |
| 1244 | Help database is corrupt or does not exist |
| 1245 | Cyclic reference on subqueries |
| 1246 | Converting column '%s' from %s to %s |
| 1247 | Reference '%s' not supported (%s) |
| 1248 | Every derived table must have its own alias |
| 1249 | Select %u was reduced during optimization |
| 1250 | Table '%s' from one of the SELECTs cannot be used in %s |
| 1251 | Client does not support authentication protocol requested by server; consider upgrading MySQL client |
| 1252 | All parts of a SPATIAL index must be NOT NULL |
| 1253 | COLLATION '%s' is not valid for CHARACTER SET '%s' |
| 1254 | Slave is already running |
| 1255 | Slave has already been stopped |
| 1256 | Uncompressed data size too large; the maximum size is %d (probably, length of uncompressed data was corrupted) |

| Error Code | Error Message |
|---|---|
| 1257 | ZLIB: Not enough memory |
| 1258 | ZLIB: Not enough room in the output buffer (probably, length of uncompressed data was corrupted) |
| 1259 | ZLIB: Input data corrupted |
| 1260 | %d line(s) were cut by GROUP_CONCAT() |
| 1261 | Row %ld doesn't contain data for all columns |
| 1262 | Row %ld was truncated; it contained more data than there were input columns |
| 1263 | Data truncated; NULL supplied to NOT NULL column '%s' at row %ld |
| 1264 | Data truncated; out of range for column '%s' at row %ld |
| 1265 | Data truncated for column '%s' at row %ld |
| 1266 | Using storage engine %s for table '%s' |
| 1267 | Illegal mix of collations (%s,%s) and (%s,%s) for operation '%s' |
| 1268 | Can't drop one or more of the requested users |
| 1269 | Can't revoke all privileges, grant for one or more of the requested users |
| 1270 | Illegal mix of collations (%s,%s), (%s,%s), (%s,%s) for operation '%s' |
| 1271 | Illegal mix of collations for operation '%s' |
| 1272 | Variable '%s' is not a variable component (can't be used as XXXX.variable_name) |
| 1273 | Unknown collation: '%s' |
| 1274 | SSL parameters in CHANGE MASTER are ignored because this MySQL slave was compiled without SSL support; they can be used later if MySQL slave with SSL is started |
| 1275 | Server is running in –secure-auth mode, but '%s'@'%s' has a password in the old format; please change the password to the new format |
| 1276 | Field or reference '%s%s%s%s%s' of SELECT #%d was resolved in SELECT #%d |
| 1277 | Incorrect parameter or combination of parameters for START SLAVE UNTIL |
| 1278 | It is recommended to use –skip-slave-start when doing step-by-step replication with START SLAVE UNTIL; otherwise, you will get problems if you get an unexpected slave's mysqld restart |
| 1279 | SQL thread is not to be started so UNTIL options are ignored |
| 1280 | Incorrect index name '%s' |
| 1281 | Incorrect catalog name '%s' |
| 1282 | Query cache failed to set size %lu; new query cache size is %lu |
| 1283 | Column '%s' cannot be part of FULLTEXT index |
| 1284 | Unknown key cache '%s' |
| 1285 | MySQL is started in –skip-name-resolve mode; you must restart it without this switch for this grant to work |
| 1286 | Unknown table engine '%s' |
| 1287 | '%s' is deprecated; use '%s' instead |

| Error Code | Error Message |
|---|---|
| 1288 | The target table %s of the %s is not updatable |
| 1289 | The '%s' feature is disabled; you need MySQL built with '%s' to have it working |
| 1290 | The MySQL server is running with the %s option so it cannot execute this statement |
| 1291 | Column '%s' has duplicated value '%s' in %s |
| 1292 | Truncated incorrect %s value: '%s' |
| 1293 | Incorrect table definition; there can be only one TIMESTAMP column with CURRENT_TIMESTAMP in DEFAULT or ON UPDATE clause |
| 1294 | Invalid ON UPDATE clause for '%s' column |
| 1295 | This command is not supported in the prepared statement protocol yet |
| 1296 | Can't create a %s from within another stored routine |
| 1297 | %s %s already exists |
| 1298 | %s %s does not exist |
| 1299 | Failed to DROP %s %s |
| 1300 | Failed to CREATE %s %s |
| 1301 | %s with no matching label: %s |
| 1302 | Redefining label %s |
| 1303 | End-label %s without match |
| 1304 | Referring to uninitialized variable %s |
| 1305 | SELECT in a stored procedure must have INTO |
| 1306 | RETURN is only allowed in a FUNCTION |
| 1307 | Statements like SELECT, INSERT, UPDATE (and others) are not allowed in a FUNCTION |
| 1308 | The update log is deprecated and replaced by the binary log; SET SQL_LOG_UPDATE has been ignored |
| 1309 | The update log is deprecated and replaced by the binary log; SET SQL_LOG_UPDATE has been translated to SET SQL_LOG_BIN |
| 1310 | Query execution was interrupted |
| 1311 | Incorrect number of arguments for %s %s; expected %u, got %u |
| 1312 | Undefined CONDITION: %s |
| 1313 | No RETURN found in FUNCTION %s |
| 1314 | FUNCTION %s ended without RETURN |
| 1315 | Cursor statement must be a SELECT |
| 1316 | Cursor SELECT must not have INTO |
| 1317 | Undefined CURSOR: %s |
| 1318 | Cursor is already open |
| 1319 | Cursor is not open |
| 1320 | Undeclared variable: %s |
| 1321 | Incorrect number of FETCH variables |
| 1322 | No data to FETCH |
| 1323 | Duplicate parameter: %s |
| 1324 | Duplicate variable: %s |
| 1325 | Duplicate condition: %s |
| 1326 | Duplicate cursor: %s |

| Error Code | Error Message |
|---|---|
| 1327 | Failed to ALTER %s %s |
| 1328 | Subselect value not supported |
| 1329 | USE is not allowed in a stored procedure |
| 1330 | Variable or condition declaration after cursor or handler declaration |
| 1331 | Cursor declaration after handler declaration |
| 1332 | Case not found for CASE statement |
| 1333 | Configuration file ‘%s’ is too big |
| 1334 | Malformed file type header in file ‘%s’ |
| 1335 | Unexpected end of file while parsing comment ‘%s’ |
| 1336 | Error while parsing parameter ‘%s’ (line: ‘%s’) |
| 1337 | Unexpected end of file while skipping unknown parameter ‘%s’ |

A

# Troubleshooting Query Problems

This appendix lists some common problems and error messages that you may encounter when executing SQL statements and what to do to solve them.

## A.1 Query-Related Issues

### A.1.1 Case Sensitivity in Searches

By default, MySQL searches are not case sensitive (although there are some character sets that are never case insensitive, such as `czech`). This means that if you search with *`col_name`* `LIKE 'a%'`, you will get all column values that start with `A` or `a`. If you want to make this search case sensitive, make sure that one of the operands is a binary string. You can do this with the `BINARY` operator. Write the condition as either `BINARY` *`col_name`* `LIKE 'a%'` or *`col_name`* `LIKE BINARY 'a%'`.

If you want a column always to be treated in case-sensitive fashion, declare it as `BINARY`. See Section 6.2.5, "`CREATE TABLE` Syntax."

Simple comparison operations (`>=, >, =, <, <=`, sorting, and grouping) are based on each character's "sort value." Characters with the same sort value (such as '`E`', '`e`', and '`é`') are treated as the same character.

If you are using Chinese data in the so-called `big5` encoding, you want to make all character columns `BINARY`. This works because the sorting order of `big5` encoding characters is based on the order of ASCII codes. As of MySQL 4.1, you can explicitly declare that a column should use the `big5` character set:

```
CREATE TABLE t (name CHAR(40) CHARACTER SET big5);
```

## A.1.2 Problems Using DATE Columns

The format of a `DATE` value is `'YYYY-MM-DD'`. According to standard SQL, no other format is allowed. You should use this format in `UPDATE` expressions and in the `WHERE` clause of `SELECT` statements. For example:

```
mysql> SELECT * FROM tbl_name WHERE date >= '2003-05-05';
```

As a convenience, MySQL automatically converts a date to a number if the date is used in a numeric context (and vice versa). It is also smart enough to allow a "relaxed" string form when updating and in a `WHERE` clause that compares a date to a `TIMESTAMP`, `DATE`, or `DATETIME` column. ("Relaxed form" means that any punctuation character may be used as the separator between parts. For example, `'2004-08-15'` and `'2004#08#15'` are equivalent.) MySQL can also convert a string containing no separators (such as `'20040815'`), provided it makes sense as a date.

The special date `'0000-00-00'` can be stored and retrieved as `'0000-00-00'`. When using a `'0000-00-00'` date through Connector/ODBC, it is automatically converted to `NULL` in Connector/ODBC 2.50.12 and above, because ODBC can't handle this kind of date.

Because MySQL performs the conversions described above, the following statements work:

```
mysql> INSERT INTO tbl_name (idate) VALUES (19970505);
mysql> INSERT INTO tbl_name (idate) VALUES ('19970505');
mysql> INSERT INTO tbl_name (idate) VALUES ('97-05-05');
mysql> INSERT INTO tbl_name (idate) VALUES ('1997.05.05');
mysql> INSERT INTO tbl_name (idate) VALUES ('1997 05 05');
mysql> INSERT INTO tbl_name (idate) VALUES ('0000-00-00');

mysql> SELECT idate FROM tbl_name WHERE idate >= '1997-05-05';
mysql> SELECT idate FROM tbl_name WHERE idate >= 19970505;
mysql> SELECT MOD(idate,100) FROM tbl_name WHERE idate >= 19970505;
mysql> SELECT idate FROM tbl_name WHERE idate >= '19970505';
```

However, the following will not work:

```
mysql> SELECT idate FROM tbl_name WHERE STRCMP(idate,'20030505')=0;
```

`STRCMP()` is a string function, so it converts `idate` to a string in `'YYYY-MM-DD'` format and performs a string comparison. It does not convert `'20030505'` to the date `'2003-05-05'` and perform a date comparison.

The MySQL server packs dates for storage, so it can't store a given date if the date would not fit onto the result buffer. MySQL does very limited checking of whether the date is correct. If you store an incorrect date, such as `'2004-2-31'`, MySQL stores it as given. The rules for accepting a date are:

- If MySQL can store and retrieve a given date as given, the date is accepted for `DATE` and `DATETIME` columns even if it is not strictly legal.
- Day values from 0 to 31 are accepted for any date. This makes it very convenient for Web applications where you ask year, month, and day in three different fields.
- The day or month value may be zero. This is convenient if you want to store a birthdate in a `DATE` column and you know only part of the date.

If the date cannot be converted to any reasonable value, a `0` is stored in the `DATE` column, which will be retrieved as `'0000-00-00'`. This is both a speed and a convenience issue. We believe that the database server's responsibility is to retrieve the same date you stored (even if the data was not logically correct in all cases). We think it is up to the application and not the server to check the dates.

## A.1.3 Problems with `NULL` Values

The concept of the `NULL` value is a common source of confusion for newcomers to SQL, who often think that `NULL` is the same thing as an empty string `''`. This is not the case. For example, the following statements are completely different:

```
mysql> INSERT INTO my_table (phone) VALUES (NULL);
mysql> INSERT INTO my_table (phone) VALUES ('');
```

Both statements insert a value into the `phone` column, but the first inserts a `NULL` value and the second inserts an empty string. The meaning of the first can be regarded as "phone number is not known" and the meaning of the second can be regarded as "the person is known to have no phone, and thus no phone number."

To help with `NULL` handling, you can use the `IS NULL` and `IS NOT NULL` operators and the `IFNULL()` function.

In SQL, the `NULL` value is never true in comparison to any other value, even `NULL`. 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. All columns in the following example return `NULL`:

```
mysql> SELECT NULL, 1+NULL, CONCAT('Invisible',NULL);
```

If you want to search for column values that are `NULL`, you cannot use an *`expr`* `= NULL` test. The following statement returns no rows, because *`expr`* `= NULL` is never true for any expression:

```
mysql> SELECT * FROM my_table WHERE phone = NULL;
```

To look for NULL values, you must use the IS NULL test. The following statements show how to find the NULL phone number and the empty phone number:

```
mysql> SELECT * FROM my_table WHERE phone IS NULL;
mysql> SELECT * FROM my_table WHERE phone = '';
```

You can add an index on a column that can have NULL values if you are using MySQL 3.23.2 or newer and are using the MyISAM, InnoDB, or BDB storage engine. As of MySQL 4.0.2, the MEMORY storage engine also supports NULL values in indexes. Otherwise, you must declare an indexed column NOT NULL and you cannot insert NULL into the column.

When reading data with LOAD DATA INFILE, empty or missing columns are updated with ''. If you want a NULL value in a column, you should use \N in the data file. The literal word "NULL" may also be used under some circumstances. See Section 6.1.5, "LOAD DATA INFILE Syntax."

When using DISTINCT, GROUP BY, or ORDER BY, all NULL values are regarded as equal.

When using ORDER BY, NULL values are presented first, or last if you specify DESC to sort in descending order. Exception: In MySQL 4.0.2 through 4.0.10, NULL values sort first regardless of sort order.

Aggregate (summary) functions such as COUNT(), MIN(), and SUM() ignore NULL values. The exception to this is COUNT(*), which counts rows and not individual column values. For example, the following statement produces two counts. The first is a count of the number of rows in the table, and the second is a count of the number of non-NULL values in the age column:

```
mysql> SELECT COUNT(*), COUNT(age) FROM person;
```

For some column types, MySQL handles NULL values specially. If you insert NULL into a TIMESTAMP column, the current date and time are inserted. If you insert NULL into an integer column that has the AUTO_INCREMENT attribute, the next number in the sequence is inserted.

## A.1.4 Problems with Column Aliases

You can use an alias to refer to a column in GROUP BY, ORDER BY, or HAVING clauses. Aliases can also be used to give columns better names:

```
SELECT SQRT(a*b) AS route FROM tbl_name GROUP BY route HAVING route > 0;
SELECT id, COUNT(*) AS cnt FROM tbl_name GROUP BY id HAVING cnt > 0;
SELECT id AS 'Customer identity' FROM tbl_name;
```

Standard SQL doesn't allow you to refer to a column alias in a WHERE clause. This is because when the WHERE code is executed, the column value may not yet be determined. For example, the following query is illegal:

```
SELECT id, COUNT(*) AS cnt FROM tbl_name WHERE cnt > 0 GROUP BY id;
```

The WHERE statement is executed to determine which rows should be included in the GROUP BY part, whereas HAVING is used to decide which rows from the result set should be used.

## A.1.5 Rollback Failure for Non-Transactional Tables

If you receive the following message when trying to perform a `ROLLBACK`, it means that one or more of the tables you used in the transaction do not support transactions:

```
Warning: Some non-transactional changed tables couldn't be rolled back
```

These non-transactional tables will not be affected by the `ROLLBACK` statement.

If you were not deliberately mixing transactional and non-transactional tables within the transaction, the most likely cause for this message is that a table you thought was transactional actually is not. This can happen if you try to create a table using a transactional storage engine that is not supported by your `mysqld` server (or that was disabled with a startup option). If `mysqld` doesn't support a storage engine, it will instead create the table as a `MyISAM` table, which is non-transactional.

You can check the table type for a table by using either of these statements:

```
SHOW TABLE STATUS LIKE 'tbl_name';
SHOW CREATE TABLE tbl_name;
```

See Section 6.5.3.17, "`SHOW TABLE STATUS` Syntax," and Section 6.5.3.6, "`SHOW CREATE TABLE` Syntax."

You can check which storage engines your `mysqld` server supports by using this statement:

```
SHOW ENGINES;
```

Before MySQL 4.1.2, `SHOW ENGINES` is unavailable. Use the following statement instead and check the value of the variable that is associated with the storage engine in which you are interested:

```
SHOW VARIABLES LIKE 'have_%';
```

For example, to determine whether the `InnoDB` storage engine is available, check the value of the `have_innodb` variable.

See Section 6.5.3.8, "`SHOW ENGINES` Syntax," and Section 6.5.3.19, "`SHOW VARIABLES` Syntax."

## A.1.6 Deleting Rows from Related Tables

MySQL does not support subqueries prior to Version 4.1, or the use of more than one table in the `DELETE` statement prior to Version 4.0. If your version of MySQL does not support subqueries or multiple-table `DELETE` statements, you can use the following approach to delete rows from two related tables:

1. `SELECT` the rows based on some `WHERE` condition in the main table.
2. `DELETE` the rows in the main table based on the same condition.
3. `DELETE FROM` *`related_table`* `WHERE` *`related_column`* `IN (`*`selected_rows`*`)`.

If the total length of the `DELETE` statement for *`related_table`* is more than 1MB (the default value of the `max_allowed_packet` system variable), you should split it into smaller parts and execute multiple `DELETE` statements. You will probably get the fastest `DELETE` by specifying only 100 to 1,000 *`related_column`* values per statement if the *`related_column`* is indexed. If the *`related_column`* isn't indexed, the speed is independent of the number of arguments in the `IN` clause.

## A.1.7 Solving Problems with No Matching Rows

If you have a complicated query that uses many tables but that doesn't return any rows, you should use the following procedure to find out what is wrong:

1. Test the query with `EXPLAIN` to check whether you can find something that is obviously wrong.
2. Select only those columns that are used in the `WHERE` clause.
3. Remove one table at a time from the query until it returns some rows. If the tables are large, it's a good idea to use `LIMIT 10` with the query.
4. Issue a `SELECT` for the column that should have matched a row against the table that was last removed from the query.
5. If you are comparing `FLOAT` or `DOUBLE` columns with numbers that have decimals, you can't use equality (=) comparisons. This problem is common in most computer languages because not all floating-point values can be stored with exact precision. In some cases, changing the `FLOAT` to a `DOUBLE` will fix this. See Section A.1.8, "Problems with Floating-Point Comparisons."
6. If you still can't figure out what's wrong, create a minimal test that can be run with `mysql test < query.sql` that shows your problems. You can create a test file by dumping the tables with `mysqldump --quick` *`db_name tbl_name_1`* `...` *`tbl_name_n`* `> query.sql`. Open the file in an editor, remove some insert lines (if there are more than needed to demonstrate the problem), and add your `SELECT` statement at the end of the file.

   Verify that the test file demonstrates the problem by executing these commands:

   ```
   shell> mysqladmin create test2
   shell> mysql test2 < query.sql
   ```

   Post the test file using `mysqlbug` to the general MySQL mailing list. See Section 1.7.1.1, "The MySQL Mailing Lists."

## A.1.8 Problems with Floating-Point Comparisons

Floating-point numbers sometimes cause confusion because they are not stored as exact values inside computer architecture. What you can see on the screen usually is not the exact value of the number. The column types `FLOAT`, `DOUBLE`, and `DECIMAL` are such. `DECIMAL` columns store values with exact precision because they are represented as strings, but calculations on DECIMAL values may be done using floating-point operations.

The following example demonstrates the problem. It shows that even for the DECIMAL column type, calculations that are done using floating-point operations are subject to floating-point error.

```
mysql> CREATE TABLE t1 (i INT, d1 DECIMAL(9,2), d2 DECIMAL(9,2));
mysql> INSERT INTO t1 VALUES (1, 101.40, 21.40), (1, -80.00, 0.00),
    -> (2, 0.00, 0.00), (2, -13.20, 0.00), (2, 59.60, 46.40),
    -> (2, 30.40, 30.40), (3, 37.00, 7.40), (3, -29.60, 0.00),
    -> (4, 60.00, 15.40), (4, -10.60, 0.00), (4, -34.00, 0.00),
    -> (5, 33.00, 0.00), (5, -25.80, 0.00), (5, 0.00, 7.20),
    -> (6, 0.00, 0.00), (6, -51.40, 0.00);

mysql> SELECT i, SUM(d1) AS a, SUM(d2) AS b
    -> FROM t1 GROUP BY i HAVING a <> b;
+------+--------+-------+
| i    | a      | b     |
+------+--------+-------+
|    1 |  21.40 | 21.40 |
|    2 |  76.80 | 76.80 |
|    3 |   7.40 |  7.40 |
|    4 |  15.40 | 15.40 |
|    5 |   7.20 |  7.20 |
|    6 | -51.40 |  0.00 |
+------+--------+-------+
```

The result is correct. Although the first five records look like they shouldn't pass the comparison test (the values of a and b do not appear to be different), they may do so because the difference between the numbers shows up around the tenth decimal or so, depending on computer architecture.

The problem cannot be solved by using ROUND() or similar functions, because the result is still a floating-point number:

```
mysql> SELECT i, ROUND(SUM(d1), 2) AS a, ROUND(SUM(d2), 2) AS b
    -> FROM t1 GROUP BY i HAVING a <> b;
+------+--------+-------+
| i    | a      | b     |
+------+--------+-------+
|    1 |  21.40 | 21.40 |
|    2 |  76.80 | 76.80 |
|    3 |   7.40 |  7.40 |
|    4 |  15.40 | 15.40 |
|    5 |   7.20 |  7.20 |
|    6 | -51.40 |  0.00 |
+------+--------+-------+
```

This is what the numbers in column a look like when displayed with more decimal places:

```
mysql> SELECT i, ROUND(SUM(d1), 2)*1.0000000000000000 AS a,
    -> ROUND(SUM(d2), 2) AS b FROM t1 GROUP BY i HAVING a <> b;
+------+---------------------+-------+
| i    | a                   | b     |
+------+---------------------+-------+
|    1 |  21.3999999999999986 | 21.40 |
|    2 |  76.7999999999999972 | 76.80 |
|    3 |   7.4000000000000004 |  7.40 |
|    4 |  15.4000000000000004 | 15.40 |
|    5 |   7.2000000000000002 |  7.20 |
|    6 | -51.3999999999999986 |  0.00 |
+------+---------------------+-------+
```

Depending on your computer architecture, you may or may not see similar results. Different CPUs may evaluate floating-point numbers differently. For example, on some machines you may get the "correct" results by multiplying both arguments by 1, as the following example shows.

**Warning:** Never use this method in your applications. It is not an example of a trustworthy method!

```
mysql> SELECT i, ROUND(SUM(d1), 2)*1 AS a, ROUND(SUM(d2), 2)*1 AS b
    -> FROM t1 GROUP BY i HAVING a <> b;
+------+--------+------+
| i    | a      | b    |
+------+--------+------+
|    6 | -51.40 | 0.00 |
+------+--------+------+
```

The reason that the preceding example seems to work is that on the particular machine where the test was done, CPU floating-point arithmetic happens to round the numbers to the same value. However, there is no rule that any CPU should do so, so this method cannot be trusted.

The correct way to do floating-point number comparison is to first decide on an acceptable tolerance for differences between the numbers and then do the comparison against the tolerance value. For example, if we agree that floating-point numbers should be regarded the same if they are the same within a precision of one in ten thousand (0.0001), the comparison should be written to find differences larger than the tolerance value:

```
mysql> SELECT i, SUM(d1) AS a, SUM(d2) AS b FROM t1
    -> GROUP BY i HAVING ABS(a - b) > 0.0001;
+------+--------+------+
| i    | a      | b    |
+------+--------+------+
|    6 | -51.40 | 0.00 |
+------+--------+------+
1 row in set (0.00 sec)
```

Conversely, to get rows where the numbers are the same, the test should find differences within the tolerance value:

```
mysql> SELECT i, SUM(d1) AS a, SUM(d2) AS b FROM t1
    -> GROUP BY i HAVING ABS(a - b) <= 0.0001;
+------+-------+-------+
| i    | a     | b     |
+------+-------+-------+
|    1 | 21.40 | 21.40 |
|    2 | 76.80 | 76.80 |
|    3 |  7.40 |  7.40 |
|    4 | 15.40 | 15.40 |
|    5 |  7.20 |  7.20 |
+------+-------+-------+
```

## A.2 Optimizer-Related Issues

MySQL uses a cost-based optimizer to determine the best way to resolve a query. In many cases, MySQL can calculate the best possible query plan, but sometimes MySQL doesn't have enough information about the data at hand and has to make "educated" guesses about the data.

For the cases when MySQL does not do the "right" thing, tools that you have available to help MySQL are:

- Use the `EXPLAIN` statement to get information about how MySQL will process a query. To use it, just add the keyword `EXPLAIN` to the front of your `SELECT` statement:

  ```
  mysql> EXPLAIN SELECT * FROM t1, t2 WHERE t1.i = t2.i;
  ```

  `EXPLAIN` is discussed in more detail in the *MySQL Administrator's Guide*.
- Use `ANALYZE TABLE` *`tbl_name`* to update the key distributions for the scanned table. See Section 6.5.2.1, "`ANALYZE TABLE` Syntax."
- Use `FORCE INDEX` for the scanned table to tell MySQL that table scans are very expensive compared to using the given index. See Section 6.1.7, "`SELECT` Syntax."

  ```
  SELECT * FROM t1, t2 FORCE INDEX (index_for_column)
  WHERE t1.col_name=t2.col_name;
  ```

  `USE INDEX` and `IGNORE INDEX` may also be useful.
- Global and table-level `STRAIGHT_JOIN`. See Section 6.1.7, "`SELECT` Syntax."
- You can tune global or thread-specific system variables. For example, start `mysqld` with the `--max-seeks-for-key=1000` option or use `SET max_seeks_for_key=1000` to tell the optimizer to assume that no key scan will cause more than 1,000 key seeks.

## A.3 Table Definition-Related Issues

### A.3.1 Problems with `ALTER TABLE`

`ALTER TABLE` changes a table to the current character set. If you get a duplicate-key error during `ALTER TABLE`, the cause is either that the new character set maps two keys to the same value or that the table is corrupted. In the latter case, you should run `REPAIR TABLE` on the table.

If `ALTER TABLE` dies with the following error, the problem may be that MySQL crashed during an earlier `ALTER TABLE` operation and there is an old table named `A-`*`xxx`* or `B-`*`xxx`* lying around:

```
Error on rename of './database/name.frm'
to './database/B-xxx.frm' (Errcode: 17)
```

In this case, go to the MySQL data directory and delete all files that have names starting with `A-` or `B-`. (You may want to move them elsewhere instead of deleting them.)

`ALTER TABLE` works in the following way:

- Create a new table named `A-`*`xxx`* with the requested structural changes.
- Copy all rows from the original table to `A-`*`xxx`*.
- Rename the original table to `B-`*`xxx`*.
- Rename `A-`*`xxx`* to your original table name.
- Delete `B-`*`xxx`*.

If something goes wrong with the renaming operation, MySQL tries to undo the changes. If something goes seriously wrong (although this shouldn't happen), MySQL may leave the old table as `B-xxx`. A simple rename of the table files at the system level should get your data back.

If you use `ALTER TABLE` on a transactional table or if you are using Windows or OS/2, `ALTER TABLE` will `UNLOCK` the table if you had done a `LOCK TABLE` on it. This is because `InnoDB` and these operating systems cannot drop a table that is in use.

### A.3.2 How to Change the Order of Columns in a Table

First, consider whether you really need to change the column order in a table. The whole point of SQL is to abstract the application from the data storage format. You should always specify the order in which you wish to retrieve your data. The first of the following statements returns columns in the order *`col_name1`*, *`col_name2`*, *`col_name3`*, whereas the second returns them in the order *`col_name1`*, *`col_name3`*, *`col_name2`*:

```
mysql> SELECT col_name1, col_name2, col_name3 FROM tbl_name;
mysql> SELECT col_name1, col_name3, col_name2 FROM tbl_name;
```

If you decide to change the order of table columns anyway, you can do so as follows:

1. Create a new table with the columns in the new order.
2. Execute this statement:

   ```
   mysql> INSERT INTO new_table
       -> SELECT columns-in-new-order FROM old_table;
   ```

3. Drop or rename `old_table`.
4. Rename the new table to the original name:

   ```
   mysql> ALTER TABLE new_table RENAME old_table;
   ```

`SELECT *` is quite suitable for testing queries. However, in an application, you should *never* rely on using `SELECT *` and retrieving the columns based on their position. The order and position in which columns are returned will not remain the same if you add, move, or delete columns. A simple change to your table structure will cause your application to fail.

## A.3.3 TEMPORARY TABLE Problems

The following list indicates limitations on the use of `TEMPORARY` tables:

- A `TEMPORARY` table can only be of type `HEAP`, `ISAM`, `MyISAM`, `MERGE`, or `InnoDB`.
- You cannot refer to a `TEMPORARY` table more than once in the same query. For example, the following does not work:

  ```
  mysql> SELECT * FROM temp_table, temp_table AS t2;
  ERROR 1137: Can't reopen table: 'temp_table'
  ```

- The `SHOW TABLES` statement does not list `TEMPORARY` tables.
- You cannot use `RENAME` to rename a `TEMPORARY` table. However, you can use `ALTER TABLE` instead:

  ```
  mysql> ALTER TABLE orig_name RENAME new_name;
  ```

B

# MySQL Regular Expressions

A regular expression is a powerful way of specifying a pattern for a complex search.

MySQL uses Henry Spencer's implementation of regular expressions, which is aimed at conformance with POSIX 1003.2. MySQL uses the extended version to support pattern-matching operations performed with the `REGEXP` operator in SQL statements.

This appendix is a summary, with examples, of the special characters and constructs that can be used in MySQL for `REGEXP` operations. It does not contain all the details that can be found in Henry Spencer's `regex(7)` manual page. That manual page is included in MySQL source distributions, in the `regex.7` file under the `regex` directory.

A regular expression describes a set of strings. The simplest regular expression is one that has no special characters in it. For example, the regular expression `hello` matches `hello` and nothing else.

Non-trivial regular expressions use certain special constructs so that they can match more than one string. For example, the regular expression `hello|word` matches either the string `hello` or the string `word`.

As a more complex example, the regular expression `B[an]*s` matches any of the strings `Bananas`, `Baaaaas`, `Bs`, and any other string starting with a `B`, ending with an `s`, and containing any number of `a` or `n` characters in between.

A regular expression for the `REGEXP` operator may use any of the following special characters and constructs:

- `^`

  Match the beginning of a string.

  ```
  mysql> SELECT 'fo\nfo' REGEXP '^fo$';                    -> 0
  mysql> SELECT 'fofo' REGEXP '^fo';                       -> 1
  ```

- `$`

  Match the end of a string.

  ```
  mysql> SELECT 'fo\no' REGEXP '^fo\no$';                  -> 1
  mysql> SELECT 'fo\no' REGEXP '^fo$';                     -> 0
  ```

- `.`

  Match any character (including carriage return and newline).

  ```
  mysql> SELECT 'fofo' REGEXP '^f.*$';                  -> 1
  mysql> SELECT 'fo\r\nfo' REGEXP '^f.*$';              -> 1
  ```

- `a*`

  Match any sequence of zero or more a characters.

  ```
  mysql> SELECT 'Ban' REGEXP '^Ba*n';                   -> 1
  mysql> SELECT 'Baaan' REGEXP '^Ba*n';                 -> 1
  mysql> SELECT 'Bn' REGEXP '^Ba*n';                    -> 1
  ```

- `a+`

  Match any sequence of one or more a characters.

  ```
  mysql> SELECT 'Ban' REGEXP '^Ba+n';                   -> 1
  mysql> SELECT 'Bn' REGEXP '^Ba+n';                    -> 0
  ```

- `a?`

  Match either zero or one a character.

  ```
  mysql> SELECT 'Bn' REGEXP '^Ba?n';                    -> 1
  mysql> SELECT 'Ban' REGEXP '^Ba?n';                   -> 1
  mysql> SELECT 'Baan' REGEXP '^Ba?n';                  -> 0
  ```

- `de|abc`

  Match either of the sequences `de` or `abc`.

  ```
  mysql> SELECT 'pi' REGEXP 'pi|apa';                   -> 1
  mysql> SELECT 'axe' REGEXP 'pi|apa';                  -> 0
  mysql> SELECT 'apa' REGEXP 'pi|apa';                  -> 1
  mysql> SELECT 'apa' REGEXP '^(pi|apa)$';              -> 1
  mysql> SELECT 'pi' REGEXP '^(pi|apa)$';               -> 1
  mysql> SELECT 'pix' REGEXP '^(pi|apa)$';              -> 0
  ```

- `(abc)*`

  Match zero or more instances of the sequence `abc`.

  ```
  mysql> SELECT 'pi' REGEXP '^(pi)*$';                  -> 1
  mysql> SELECT 'pip' REGEXP '^(pi)*$';                 -> 0
  mysql> SELECT 'pipi' REGEXP '^(pi)*$';                -> 1
  ```

- `{1}`, `{2,3}`

  `{n}` or `{m,n}` notation provides a more general way of writing regular expressions that match many occurrences of the previous atom (or "piece") of the pattern. *m* and *n* are integers.

  - `a*`

    Can be written as `a{0,}`.

  - `a+`

    Can be written as `a{1,}`.

  - `a?`

    Can be written as `a{0,1}`.

  To be more precise, `a{n}` matches exactly *n* instances of `a`. `a{n,}` matches *n* or more instances of `a`. `a{m,n}` matches *m* through *n* instances of `a`, inclusive.

  *m* and *n* must be in the range from `0` to `RE_DUP_MAX` (default 255), inclusive. If both *m* and *n* are given, *m* must be less than or equal to *n*.

```
mysql> SELECT 'abcde' REGEXP 'a[bcd]{2}e';          -> 0
mysql> SELECT 'abcde' REGEXP 'a[bcd]{3}e';          -> 1
mysql> SELECT 'abcde' REGEXP 'a[bcd]{1,10}e';       -> 1
```

- `[a-dX]`, `[^a-dX]`

  Matches any character that is (or is not, if `^` is used) either `a`, `b`, `c`, `d` or `X`. A `-` character between two other characters forms a range that matches all characters from the first character to the second. For example, `[0-9]` matches any decimal digit. To include a literal `]` character, it must immediately follow the opening bracket `[`. To include a literal `-` character, it must be written first or last. Any character that does not have a special defined meaning inside a `[]` pair matches only itself.

```
mysql> SELECT 'aXbc' REGEXP '[a-dXYZ]';             -> 1
mysql> SELECT 'aXbc' REGEXP '^[a-dXYZ]$';           -> 0
mysql> SELECT 'aXbc' REGEXP '^[a-dXYZ]+$';          -> 1
mysql> SELECT 'aXbc' REGEXP '^[^a-dXYZ]+$';         -> 0
mysql> SELECT 'gheis' REGEXP '^[^a-dXYZ]+$';        -> 1
mysql> SELECT 'gheisa' REGEXP '^[^a-dXYZ]+$';       -> 0
```

- `[.characters.]`

  Within a bracket expression (written using `[` and `]`), matches the sequence of characters of that collating element. *`characters`* is either a single character or a character name like `newline`. You can find the full list of character names in the `regexp/cname.h` file.

```
mysql> SELECT '~' REGEXP '[[.~.]]';                 -> 1
mysql> SELECT '~' REGEXP '[[.tilde.]]';             -> 1
```

- `[=character_class=]`

  Within a bracket expression (written using `[` and `]`), `[=character_class=]` represents an equivalence class. It matches all characters with the same collation value, including itself. For example, if `o` and `(+)` are the members of an equivalence class, then `[[=o=]]`, `[[=(+)=]]`, and `[o(+)]` are all synonymous. An equivalence class may not be used as an endpoint of a range.

- `[:character_class:]`

  Within a bracket expression (written using `[` and `]`), `[:character_class:]` represents a character class that matches all characters belonging to that class. The standard class names are:

  | | |
  |---|---|
  | `alnum` | Alphanumeric characters |
  | `alpha` | Alphabetic characters |
  | `blank` | Whitespace characters |
  | `cntrl` | Control characters |
  | `digit` | Digit characters |
  | `graph` | Graphic characters |
  | `lower` | Lowercase alphabetic characters |
  | `print` | Graphic or space characters |
  | `punct` | Punctuation characters |
  | `space` | Space, tab, newline, and carriage return |
  | `upper` | Uppercase alphabetic characters |
  | `xdigit` | Hexadecimal digit characters |

  These stand for the character classes defined in the `ctype(3)` manual page. A particular locale may provide other class names. A character class may not be used as an endpoint of a range.

  ```
  mysql> SELECT 'justalnums' REGEXP '[[:alnum:]]+';       -> 1
  mysql> SELECT '!!' REGEXP '[[:alnum:]]+';               -> 0
  ```

- `[[:<:]]`, `[[:>:]]`

  These markers stand for word boundaries. They match the beginning and end of words, respectively. A word is a sequence of word characters that is not preceded by or followed by word characters. A word character is an alphanumeric character in the `alnum` class or an underscore (_).

  ```
  mysql> SELECT 'a word a' REGEXP '[[:<:]]word[[:>:]]';   -> 1
  mysql> SELECT 'a xword a' REGEXP '[[:<:]]word[[:>:]]';  -> 0
  ```

To use a literal instance of a special character in a regular expression, precede it by two backslash (\) characters. The MySQL parser interprets one of the backslashes, and the regular expression library interprets the other. For example, to match the string `1+2` that contains the special + character, only the last of the following regular expressions is the correct one:

```
mysql> SELECT '1+2' REGEXP '1+2';                        -> 0
mysql> SELECT '1+2' REGEXP '1\+2';                       -> 0
mysql> SELECT '1+2' REGEXP '1\\+2';                      -> 1
```

# Index