SQL Reference

This chapter gives an overview of the SQL operators, functions, and commands available under MySQL 5.0. My goal in organizing the information in this chapter was to give you, dear reader, a compact overview of the most important and useful syntactic variants.

This chapter is no substitute for the MySQL documentation, which provides not only the MySQL source code, but also the best, most complete, and most up-to-date reference for MySQL. No book could supersede this online reference. The great advantage of the MySQL documentation, its huge amount of information, is also its greatest drawback: important commands and syntax variants get lost amid the countless details, which for 90 percent of the cases are irrelevant. See `http://dev.mysql.com/doc/mysql/en/index.html`.

Please note that MySQL boasts countless extensions as well as, alas, certain shortcomings with respect to the ANSI-SQL/92 standard.

Syntax

We begin with a brief section describing the syntax of object names, character strings, dates and times, and binary data.

Object Names

Names of objects—databases, tables, columns, etc.—can be at most 64 characters in length. Permitted characters are all the alphanumeric characters of the character set used by MySQL as well as the characters _ and $. For practical reasons, however, it makes sense to restrict oneself to the alphanumeric ASCII characters together with the underscore character. There are two reasons for this:

- The coding of special characters depends on the character set. If the client and server are not in agreement on the character set, then access to objects might become problematic.
- The names of databases and tables can be stored in files, and it is not MySQL, but the operating system that is responsible for the naming rules for files. This can be yet another source of conflict (especially if databases must be exchanged among various operating systems).

Names with special characters and reserved words: Object names are normally not permitted to be the same as reserved SQL key words (*select*, *from*, etc.). Also, most special characters are not permitted in a name (*-!%*, etc.). Both restrictions can be gotten around, however, by putting the name in backward single quotes:

```
CREATE TABLE `special name` (`from` INT, `a!%` INT)
```

Compound names: Table names that do not refer to the current database must have the database name prefixed to them. Likewise, the name of a column must be extended by the name of the table and that of the database if the column name alone fails to provide a unique identification (such as in queries in which several like-named columns in different tables appear):

Table names: *tablename* or *db.tablename*

Column names: *colname* or *tblname.colname* or *dbname.tblname.colname*

Case Sensitivity

The following objects are listed according to whether they exhibit case sensitivity:

Case Sensitivity: Database names (except under Windows), table names (except under Windows), alias names, variable names through MySQL 4.1.

No Case Sensitivity: SQL commands and functions, column names, index names, variable names since MySQL 5.0.

Under Windows, MySQL is flexible with respect to case in the naming of databases and tables. The reason is that the operating system does not distinguish case in the naming of directories and files. Note, however, that case must be consistent within an SQL command. The following command will not function properly: *SELECT * FROM authors WHERE Authors.authName = "xxx".*

Since MySQL 4.0, MySQL under Windows uses exclusively lowercase names in the creation of new databases and tables (regardless of how it is written in the *CREATE* command). This should simplify the migration of databases from Windows to Unix/Linux. This automatic transformation is the result of the option `lower_case_table_names`, which is set to 1 under Windows by default.

Character Strings

Character strings can be enclosed in single or double quotes. The following two expressions are equivalent in MySQL, though only the single-quote variant conforms to the ANSI-SQL/92 standard.

```
'character string'
"character string"
```

If a quotation mark should happen to be part of the character string, then there are various ways of expressing this:

```
"abc'abc"   means   abc'abc
"abc""abc"  means   abc"abc
"abc\'abc"  means   abc'abc
"abc\"abc"  means   abc"abc
'abc"abc'   means   abc"abc
'abc''abc'  means   abc'abc
'abc\"abc'  means   abc"abc
'abc\'abc'  means   abc'abc
```

Within a character string, the special characters provided by the prevailing character set are allowed, for example, äöüß if you are working with the default character set ISO-8859-1 (*latin1*). However, some special characters must be specially coded:

Quoting of Special Characters Within a String

`\0`	0 byte (Code 0).
`\b`	Backspace character (Code 8).
`\t`	Tab character (Code 9).
`\n`	Newline character (Code 10).
`\r`	Carriage return character (Code 13).
`\"`	Double quote (Code 34).
`\'`	Single quote (Code 39).
`\\`	Backslash (Code 92).

If *x* is not one of the above-mentioned special characters, then *\x* simply returns the character *x*. Even if a character string is to be stored as a BLOB (binary object), then the 0 character as well as the single quote, double quote, and backslash must be given in the form *\0*, *\'*, *\"*, and \\.

Instead of indicating special characters in character strings or BLOBs by the backslash escape character, it is often easier simply to specify the entire object using hexadecimal notation. MySQL accepts hex codes of arbitrary length in SQL commands in the following form:
`0x4142434445464748494a`.

However, MySQL is incapable of returning the result of a query in this form. (If you are working with PHP, then that programming language offers a convenient function for this purpose: *bin2hex*.)

Tip If two character strings are to be concatenated, then you must use the function *CONCAT*. (The operators + and || from other SQL dialects or programming languages will not serve the purpose.) In general, MySQL provides a broad range of functions for working with character strings.

Character Set and Sort Order

- The character string and sort order (*collation*) will be set independently of each other.
- Each table, and indeed each column within a table, can have its own character set and its own sort order.
- Unicode is now a possible choice for the character set (in formats UTF8 and UCS2 = UTF16).

In SQL commands one can specify the character set for every string. For this, you can use the function *CONVERT* or the cast operator *_characterset*. Here are two equivalent examples, demonstrating the internal Unicode encoding in the format UTF8:

```
SELECT HEX(CONVERT('ABCäöü' USING utf8))
  414243C3A4C3B6C3BC
SELECT HEX(_utf8 'ABCäöü')
  414243C3A4C3B6C3BC
```

In many cases, it can be necessary to specify the sort order as well as the character set. (This determines which characters are considered equivalent and how character strings are to be sorted.) For this, one has the syntax *_characterset 'abc' COLLATE collname*. Here is an example:

```
SELECT  _latin1 'a'  =  _latin1 'ä'
  0
SELECT  _latin1 'a' COLLATE latin1_german1_ci  =
        _latin1 'ä' COLLATE latin1_german1_ci
  1
```

Numbers

Decimal numbers are written with a period for the decimal point and without a thousands separator (thus 27345 or 2.71828). One may also use scientific notation (6.0225e23 or 6.626e-34) for very large or very small numbers.

MySQL can also process hexadecimal numbers prefixed by 0x or in the form *x'1234'*. Depending on the context, the number is interpreted as a character string or as a 64-bit integer:

```
SELECT 0x4142434445464748494a, x'4142434445464748494a'
       ABCDEFGHIJ              ABCDEFGHIJ
SELECT 0x41 + 0
       66
```

Automatic Transformation of Numbers and Character Strings

In carrying out an operation on two different data types, MySQL makes every attempt to find a compatible data type. Integers are automatically changed into floating-point numbers if one of the operators is a floating-point number. Character strings are automatically changed into numbers if the operation involves a calculation. (If the beginning of the character string cannot be interpreted as a number, then MySQL calculates with 0.)

```
SELECT '3.14abc' + 1
4.14
```

Date and Time

MySQL represents dates as character strings of the form *2005-12-31*, and times in the form *23:59:59*. With the data type *DATETIME* both formats are simply concatenated, yielding, for example, *2005-12-31 23:59:59*.

```
USE exceptions
SELECT * FROM test_date
id  a_date      a_time    a_datetime           a_timestamp
 1  2005-12-07  09:06:29  2005-12-07 09:06:29  2005-12-07 09:06:29
```

Caution Beginning with MySQL 4.1, the default setting of *TIMESTAMP*s has changed. They are returned from the server in the format *YYYY-MM-DD HH:MM:DD*. Through MySQL 4.0, the usual form was *YYYYMMDDHHMMDD*. Add a zero if you wish to use the old format (*SELECT ts+0 FROM table*).

Version 5.0 has introduced another important change. In *DATE* and *DATETIME* columns only valid dates are now accepted. (Older versions of MySQL made only a cursory validity check, which recognized *2005-02-45* as invalid, but not *2005-02-31*.) The date *'0000-00-00'* is a special case. This value is officially permitted in MySQL as a date.

In storing dates, MySQL is quite flexible: Both numbers (e.g., *20051231*) and character strings are accepted. Hyphens are allowed in character strings, or they can simply be done without. If a year is given but no century is specified, then MySQL automatically uses the range 1970—2069. Therefore, MySQL accepts the following character strings for a *DATETIME* column: *'2005 12 31'*, *'20051231'*, *'2005.12.31'*, and *'2005&12&31'*.

Binary Data

Binary data that are to be stored in *BLOB* fields are dealt with in SQL commands like character strings. (However, there are differences in sorting.)

Binary Numbers

Since version 5.0.3, MySQL supports the data type *BIT*. Binary numbers can be written in the form *b'110010'*.

Comments

There are three ways of supplying comments in SQL commands:

```
SELECT 1   #  comment
SELECT 1   /* comment */
SELECT 1   -- comment
```

Comments that begin with # or with — (there must be a space after the —) hold until the end of the line. Comments between /* and */ can extend over several lines, as in C. Nesting is not allowed.

If you wish to write SQL code that makes use of some of the peculiarities of MySQL yet remains compatible as much as possible with other dialects, a particular variant of the comment is often useful:

```
SELECT /*! STRAIGHT_JOIN */ col FROM table ...
```

With the MySQL-specific *SELECT* extension, *STRAIGHT_JOIN* will be executed only by MySQL; all other SQL dialects will consider this a comment.

A variant of this enables differentiation among various MySQL dialects:

```
CREATE /*!32302 TEMPORARY */ TABLE ...
```

In this case, the key word *TEMPORARY* is processed only if the command is executed by MySQL 3.23.02 or a more recent version.

Semicolons at the End of SQL Commands

Neither ANSI-SQL nor the SQL dialect of MySQL allows semicolons at the end of an instruction. This syntax rule holds as well for MySQL when a single command is to be executed. However, there are cases in which semicolons are necessary:

- If you execute commands with the MySQL command interpreter (that is, the program `mysql`), you must terminate commands with a semicolon.
- In defining stored procedures and triggers, commands must be separated by semicolons.
- Since MySQL 4.1 the client library has allowed for the execution of several commands at once, and here as well the commands must be separated by semicolons. PHP also provides for the *mysqli* method *multi_query*. With other APIs a *MULTI_STATEMENT* mode must be explicitly activated, in C, for example, with *mysql_real_connect(…, CLIENT_MULTI_STATEMENTS)*.

Operators

MySQL Operators

	Arithmetic Operators
`+ - * /`	Basic calculation.
%	Modulo (remainder on integer division).
DIV	Alternative division operator (from MySQL 4.1).
MOD	Alternative modulo operator (from MySQL 4.1).
	Bit Operators
|	Binary OR.
&	Binary AND.
~	Binary negation (inverts all bits).
<<	Shifts all bits left (implies multiplication by 2*n*).
>>	Shifts all bits right (implies division by 2*n*).
	Comparison Operators
=	Equality operator.
<=>	Equality operator that permits a *NULL* comparison.
!= <>	Inequality operator.
< > <= >=	Comparison operators.
IS [NOT] NULL	*NULL* comparison.
BETWEEN	Range comparison (e.g., *x BETWEEN 1 AND 3*).
IN	Set comparison (e.g., *x IN (1, 2, 3)* or *x IN ('a', 'b', 'c')*).
NOT IN	Set comparison (e.g., *x NOT IN ('a', 'b', 'c')*).
	Pattern Comparison
[NOT] LIKE	Simple pattern comparison (e.g., *x LIKE 'm%'*).
[NOT] REGEXP	Extended pattern comparison (e.g., *x REGEXP '.*x$'*).
SOUNDS LIKE	Corresponds to *SOUNDEX(a) = SOUNDEX(b)*, since MySQL 4.1.
	Binary Comparison
BINARY	Marks the operands as binary (e.g., *BINARY x = y*).
	Logical Operators
!, *NOT*	Negation.
||, *OR*	Logical OR.
&&, *AND*	Logical AND.
XOR	Logical exclusive OR (new since MySQL 4.0).
	Casting Operators (Since MySQL 4.1)
_charset 'abc'	The character set *charset* holds for *'abc'*.
_charset 'abc'COLLATE col	The character set *charset* and sort order *col* hold for *'abc'*.

Arithmetic Operators, Bit Operators

Arithmetic operators for which one of the operands is *NULL* generally return *NULL* as result. In MySQL, a division by zero also returns the result *NULL* (in contrast to many other SQL dialects).

Comparison Operators

Comparison operators normally return 1 (corresponding to *TRUE*) or 0 (*FALSE*). Comparisons with *NULL* return *NULL*. The two exceptions are the operators <=> and *IS NULL*, which even in comparison with *NULL* return 0 or 1:

```
SELECT NULL=NULL, NULL=0
  NULL, NULL
SELECT NULL<=>NULL, NULL<=>0
  1, 0
SELECT NULL IS NULL, NULL IS 0
  1, 0
```

In the case of string comparisons with <, <=, >, and >= with *BETWEEN* (and of course with all sort operators), the character set and sort order of the affected column come into play. For strings in quotes, the character set and sort order must be given explicitly. Comparisons of strings in different character sets is not permitted.

Pattern Matching with LIKE

MySQL offers two operators for pattern matching. The simpler, and ANSI-compatible, of these is *LIKE*. As with normal character string comparison, there is no case distinction. In addition, there are two wild cards:

LIKE Search Pattern

_	Placeholder for an arbitrary character.
%	Placeholder for arbitrarily many (including 0) characters (but not for *NULL*).
_	The underscore character _.
\%	The percent sign %.

Pattern Matching with REGEXP

Considerably wider scope in the formulation of a pattern is offered by *REGEXP* and the equivalent command *RLIKE*. The relatively complicated syntax for the pattern corresponds to the Unix commands `grep` and `sed`.

REGEXP Search Patterns

	Definition of the Pattern
abc	The string *abc*.
(abc)	The string *abc* (formed into a group) .
[abc]	One of the characters *a, b, c*.
[a-z]	A character in the range a to *z*.
[^abc]	None of these characters (but any other).
.	Any character.

Continued

REGEXP Search Patterns *(Continued)*

	Appearance of the Pattern
x	The expression *x* must appear once.
x|y	The expression *x* or *y* must appear once.
x?	The expression *x* may appear once (or not at all) .
*x**	The expression *x* may appear arbitrarily often (or not at all).
x+	The expression *x* may appear arbitrarily often, but at least once.
x{n}	The expression *x* must appear exactly *n* times.
x{,n}	The expression *x* may appear at most *n* times.
x{n,}	The expression *x* must appear at least *n* times.
x{n,m}	The expression *x* must appear at least *n* and at most *m* times.
^	Placeholder for the beginning of the string.
$	Placeholder for the end of the string.
\x	Special character *x* (e.g., \\$ for $).

As with *LIKE*, there is no case distinction. Please note that *REGEXP* is successful when the search pattern is found somewhere within the character string. The search pattern is thus not required to describe the entire character string, but only a part of it. If you wish to encompass the entire character string, then you must use ^ and $ in the search pattern.

Tip The above table contains only the most important elements of *REGEXP* patterns. A complete description can be obtained under Unix/Linux with `man 7 regex`. This can also be found on the Internet, for example, at `http://linux.ctyme.com/man/alpha7.htm`.

Binary Character String Comparison

Character strings are normally compared without case being taken into consideration. Thus *'a' = 'A'* returns *1* (true). If you wish to execute a binary comparison, then you must place *BINARY* in front of one of the operands. *BINARY* is a cast operator; that is, it alters the data type of one of the operands (in this case it changes a number or character string into a binary object). *BINARY* can be used both for ordinary character string comparison and for pattern matching with *LIKE* and *REGEXP*:

```
SELECT 'a'='A', BINARY 'a' = 'A', 'a' = BINARY 'A'
       1, 0, 0
```

Logical Operators

Logical operators likewise return 0 or 1, or *NULL* if one of the operands is *NULL*. This holds also for *NOT*; that is, *NOT NULL* again returns *NULL*.

Variables and Constants

MySQL supports a variety of variable types:

Ordinary variables (user variables): Such variables are identified by a prefixed @ character. They lose their definition at the end of the MySQL session.

System and server variables: Such variables contain states or attributes of the MySQL server. These variables are identified by two prefixed @ characters (e.g., *@@binlog_cache_size*).

Many system variables exist in two versions: one specific to the current connection (e.g., *@@session.wait_timeout*) and one global for the MySQL server (e.g., *@@global.wait_timeout*, with the default value for this variable).

Structured variables: These are a special case of system variables. MySQL uses such variables at this time only for defining additional MyISAM index caches.

Local variables and parameters within stored procedures: These variables are declared within stored procedures and are valid only there. They have no identifier, and so must have a name that makes them uniquely distinguishable from table and column names.

Through MySQL 4.1, MySQL variable names were case sensitive. However, starting with MySQL 5.0, these names are case insensitive. Thus *@name*, *@Name*, and *@NAME* all denote the same variable.

Variable Assignment

The following examples show several syntax variants for assigning values to variables. Note that *SET* uses the assignment operator =, while *SELECT* uses :=. The last variant, the assignment of several columns of a record to several variables, has been possible only since MySQL 5.0, and there only if the *SELECT* command returns exactly one record.

```
SET @varname = 3
SELECT @varname := 3
SELECT @varname := COUNT(*) FROM tabelle
SELECT COUNT(*) FROM tabelle INTO @varname
SELECT title, subtitle FROM titles WHERE titleID=... INTO @t, @st
```

Evaluating and Displaying Variables

Most variables can be evaluated with *SELECT*:

```
SELECT @varname
  3
SELECT @@binlog_cache_size
@@binlog_cache_size
              32768
```

In the case of system variables, you can use *SHOW VARIABLES* in addition to *SELECT*. This command has the advantage that it can display a list of variables all at once. The @@ identifier is absent.

```
SHOW VARIABLES LIKE 'b%'
Variable_name            Value
back_log                 50
basedir                  C:\Programs\MySQL\MySQL Server 5.0
binlog_cache_size        32768
bulk_insert_buffer_size  8388608
```

Remarkably, there exist system variables that can be evaluated only with *SELECT* (e.g., *@@autocommit*) or only with *SHOW VARIABLES* (e.g., *system_time_zone*).

Global System Variables versus System Variables at the Connection Level

In its system variables, MySQL distinguishes between *SESSION* and *GLOBAL* variables. *SESSION* variables are valid only for the current session (connection), while *GLOBAL* variables hold for the entire server.

```
SELECT @@wait_timeout                -- Session (connection level)
SELECT @@session.wait_timeout        -- Session (connection level)
SELECT @@global.wait_timeout         -- Global
```

System variables can also be changed. According to whether the change is only for the current connection or should be valid globally, the following syntax variants are available. Note that with *SET*, you may omit the two @ symbols. One cannot change system variables with *SELECT*.

```
SET @@wait_timeout = 10000           -- Session (connection level)
SET @@session.wait_timeout = 10000   -- Session (connection level)
SET SESSION wait_timeout = 10000     -- Session (connection level)
SET @@global.wait_timeout = 10000    -- Global
SET GLOBAL wait_timeout = 10000      -- Global
```

Variables at the global level can be changed by users possessing the *Super* privilege. When a global variable is changed, the new value holds for all new connections, but not for those already in existence.

Changes in *SESSION* variables, on the other hand, hold only until the end of the current connection. When a new connection is made, the global default value again holds.

Tip This book does not contain a complete description of all MySQL system variables. A complete list of *GLOBAL* and *SESSION* variables can be found in the MySQL documentation. You may end up at the key word *LOCAL*, which has the same meaning in this context as *SESSION*: `http://dev.mysql.com/doc/mysql/en/system-variables.html`.

Enlightenment on the contents of the variables can be found in the MySQL documentation at the following pages: `http://dev.mysql.com/doc/mysql/en/server-parameters.html`, `http://dev.mysql.com/doc/mysql/en/set-option.html`, and `http://dev.mysql.com/doc/mysql/en/show-variables.html`.

SET PASSWORD

The command *SET* can also be used to change the connection passwords. However, *PASSWORD* is not a variable!

```
SET PASSWORD = PASSWORD('xxx')
SET PASSWORD FOR user@hostname = PASSWORD('xxx')
```

SET has some special forms, which are described later in this chapter under the heading *SET*.

Structured Variables

With many system variables there is the possibility of creating more than one instance. The MySQL documentation calls these *structured variables*. They are addressed in the form *instancename.variablenname*. Indeed, an entire group of variables can be related to an instance. (In the nomenclature of object-oriented programming, one speaks simply of objects and properties.)

At present there is only one group of structured variables in MySQL, which serves for control of cache storage for MyISAM indexes: *key_buffer_size*, *key_cache_block_size*, *key_cache_division_limit*, and *key_cache_age_threshold*. These four variables determine the size and management of RAM in which indexes of MyISAM tables are temporarily stored.

After the start of the MySQL server there automatically exists an instance of this cache object, which has the name *default*. Using *SET @@default.key_buffer_size* = … you can set the size of the default cache. The shorthand *SET @@key_buffer_size*=… automatically refers to the instance *default*.

With the following command you can create an additional cache area:

```
SET @@mycache.key_buffer_size = n
```

Now *mycache* is a new instance of the cache. With *SET @@mycache.key_cache_xxx* you can set additional properties of the cache. Then you can assign individual indexes to the new cache with the command *CACHE INDEX*. To deactivate the cache, set its size to zero:

```
SET @@mycache.key_buffer_size = 0
```

Tip The use of several index caches is useful only rarely for speed optimization. Further background information is available at `http://dev.mysql.com/doc/mysql/en/myisam-key-cache.html`, `http://dev.mysql.com/doc/mysql/en/multiple-key-caches.html`, and `http://dev.mysql.com/doc/mysql/en/structured-system-variables.html`.

Constants

Starting with version 4.1, MySQL recognizes the constants *TRUE* (1) and *FALSE* (0).

MySQL Data Types

MySQL Data Types	
	Integers
TINYINT(m)	8-bit integer (1 byte); the optional value *m* gives the desired column width in *SELECT* results (*maximum display width*), but has no influence on the allowable range of numeric values.
SMALLINT(m)	16-bit integer (2 bytes).
MEDIUMINT(m)	24- bit integer (3 bytes).
INT(m), INTEGER(m)	32- bit integer (4 bytes).
BIGINT(m)	64- bit integer (8 bytes).
	Floating-Point Numbers
FLOAT(m, d)	Floating-point number, 8-place precision (4 bytes); the optional values *m* and *d* specify the desired number of places before and after the decimal point in *SELECT* results; the values have no influence over the way the number is stored.
DOUBLE(m, d)	Floating-point number, 16-place precision (8 bytes).
R*EAL(m, d)*	Synonym for *DOUBLE*.
DECIMAL(p, s)	Fixed-point number, stored as string; arbitrary number of places (1 byte per digit + 2 bytes overhead); *p* specifies the entire number of places, where *s* is the number of places after the decimal point; default is *DECIMAL(10,0)*.
NUMERIC, DEC	Synonyms for *DECIMAL*.

Continued

MySQL Data Types *(Continued)*

	Date, Time
DATE	Date in the form *'2005-12-31'*, range *1000-01-01* to *9999-12-31* (3 bytes).
TIME	Time in the form *'23:59:59'*, range *+/-838:59:59* (3 bytes).
DATETIME	Combination of *DATE* and *TIME* in the form *'2005-12-31 23:59:59'* (8 bytes).
YEAR	Year in the range *1900–2155* (1 byte).
TIMESTAMP(m)	Date and time in the form *20051231235959* for times between 1970 und 2038 (4 bytes); the optional value *m* specifies the number of places in *SELECT* results; *m=8*, for example, has the effect that only year, month, and day are displayed.
	Character Strings
CHAR(n)	String with prescribed length, maximum 255 characters.
NATIONAL CHAR(n)	Unicode string (corresponds to *CHAR(n) CHARSET utf8* or *NCHAR(n)*).
VARCHAR(n)	String with variable length; maximum 255 characters through MySQL 4.1, and maximum 65,535 bytes since MySQL 5.0.3 for MyISAM tables, where the maximum number of characters depends on the character set.
NATIONAL VARCHAR(n)	Unicode string with variable length (corresponds to *NCHAR VARCHAR(n)*, *VARCHAR(n) CHARSET utf8*).
TINYTEXT	String with variable length, maximum 255 characters.
TEXT	String with variable length, maximum 2^{16}-1 characters.
MEDIUMTEXT	String with variable length, maximum 2^{24}-1 characters.
LONGTEXT	String with variable length, maximum 2^{32}-1 characters.
	Binary Data
TINYBLOB	Binary data with variable length, maximum 255 bytes.
BLOB	Binary data with variable length, maximum 2^{16}-1 bytes.
MEDIUMBLOB	Binary data with variable length, maximum 2^{24}-1 bytes.
LONGBLOB	Binary data with variable length, maximum 2^{32}-1 bytes.
	Geometric Data (Since MySQL 4.1)
GEOMETRY	A general geometric object; further geometric types are listed later in this chapter.
	Miscellaneous
ENUM	Enumeration of at most 65,535 strings (1 or 2 bytes).
SET	Enumeration of at most 255 strings (1 to 8 bytes).
BIT	Individual bits (since MySQL 5.0.3).
BOOL	Synonym for *TINYINT(1)*.

In the definition of columns (*CREATE TABLE*, *ALTER TABLE*), different options can be used for different columns. The following table summarizes these options. Note that not all options are suitable for all data types.

Attributes (Options) of the MySQL Data Types	
NULL	Specifies that the column may contain the value *NULL*; this setting holds by default.
NOT NULL	Forbids the value *NULL*.
DEFAULT xxx	Specifies the default value *xxx* to be used if no other input value is specified. Even if you do not specify an explicit default value, MySQL itself uses one in many cases: *NULL* when it is permitted, otherwise 0 in numeric columns, an empty string with *VARCHAR*, the date *0000-00-00* with dates, the year *0000* with *YEAR* as well as the first element of an *ENUM* enumeration.
DEFAULT CURRENT_TIMESTAMP	Has the effect on *TIMESTAMP* columns that the current time is automatically stored when new records are inserted.
ON UPDATE CURRENT_TIMESTAMP	Has the effect on *TIMESTAMP* columns that when changes are made (*UPDATE*) the current time is automatically stored.
PRIMARY KEY	Defines the column as primary key.
AUTO_INCREMENT	Results in an automatically increasing number being inserted in the column; it can be used for only one column, with integer values; moreover, the options *NOT NULL* and *PRIMARY KEY* must be specified (instead of *PRIMARY KEY*, the column can be given a *UNIQUE* index).
UNSIGNED	Integers are stored without a sign; note that calculations are also made without signs.
ZEROFILL	Integers in *SELECT* results are left-filled with zeros to fill out their length (thus five-digit numbers such as 00123 and 01234).
BINARY	With *CHAR* and *VARCHAR* columns, comparison and sort operations are executed in binary. Hence uppercase letters are sorted before lowercase ones. This is more efficient, but less practical when results are to be displayed in alphabetical order.
CHARACTER SET name, [COLLATE sort]	With strings, gives the character set and optional sort order.
COMMENT text	Stores *text* as a comment on the column (since MySQL 4.1).
SERIAL	Since MySQL 4.1 is a synonym for *BIGINT NOT NULL AUTO_INCREMENT UNIQUE*.

Command Overview (Thematic)

In the section following this one, SQL commands will be listed in alphabetical order. As a supplementary aid to orientation, we provide here a systematic overview:

Database Queries, Data Manipulation	
SELECT	Queries existing record (data search).
INSERT	Inserts new record.
REPLACE	Replaces existing record.
UPDATE	Changes existing record.
DELETE	Deletes selected records.
TRUNCATE TABLE	Deletes all records of a table.
LOAD DATA	Inserts records from a text file.
HANDLER	Reads records more efficiently than *SELECT* (since MySQL 4.0).

Transactions (Only with InnoDB Tables)	
BEGIN or *START TRANSACTION*	Begins a group of SQL commands.
COMMIT	Confirms all executed commands.
ROLLBACK	Aborts executed commands.
SAVEPOINT	Places a marker within a running transaction.

Create Databases/Tables/Views, Change Database Schema	
ALTER DATABASE	Makes changes to the database (since MySQL 4.1).
ALTER TABLE	Changes individual columns of a table, adds indexes, etc.
ALTER VIEW	Changes a view (since MySQL 5.0).
CREATE DATABASE	Creates a new database.
CREATE INDEX	Creates a new index for a table.
CREATE TABLE	Creates a new table.
CREATE VIEW	Creates a view (since MySQL 5.0).
DROP DATABASE	Deletes an entire database.
CREATE FUNCTION or *PROCEDURE*	Deletes a stored procedure (since MySQL 5.0).
DROP INDEX	Deletes an index.
DROP TABLE	Deletes an entire table.
DROP VIEW	Deletes a view (since MySQL 5.0).
RENAME TABLE	Renames a table.

Administration of Tables (General)	
ANALYZE TABLE	Returns information on internal index management.
CHECK TABLE	Tests table file for consistency errors.
FLUSH TABLES	Closes all table files and then opens them.
LOCK TABLE	Blocks tables for (write) access by other users.
OPTIMIZE TABLE	Optimizes memory use in tables.
UNLOCK TABLES	Releases tables locked with *LOCK*.

Administration of MyISAM Tables	
BACKUP TABLE	Copies table files into a backup directory.
CACHE INDEX	Assigns individual caches to table indexes.
LOAD INDEX INTO CACHE	Loads table indexes into the cache.
REPAIR TABLE	Attempts to repair defective table files.
RESTORE TABLE	Restores tables backed up with *BACKUP*.

Administration and Execution of Stored Procedures and Triggers (Since MySQL 5.0)	
ALTER FUNCTION | PROCEDURE	Changes a stored procedure.
CALL	Calls a stored procedure.
CREATE FUNCTION | PROCEDURE | TRIGGER	Creates a stored procedure or trigger.
DROP FUNCTION | PROCEDURE | TRIGGER	Deletes a stored procedure or trigger.
SHOW CREATE FUNCTION | PROCEDURE	Displays the code of a stored procedure.
SHOW FUNCTION | PROCEDURE STATUS	Returns a list of all defined stored procedures.

Information on the Database Schema, Other Administrative Information	
DESCRIBE	Same as *SHOW COLUMNS*.
EXPLAIN	Explains how a *SELECT* is executed internally.
SHOW	Displays information about databases, tables, views, fields, stored procedures, etc.

Administration, Access Privileges, etc.	
FLUSH	Empties MySQL temporary storage and reads it in again.
GRANT	Grants additional privileges.
KILL	Ends a process.
REVOKE	Restricts access privileges.
RESET	Deletes the query cache or logging files.
SET	Changes the content of MySQL system variables.
SHOW	Displays the MySQL status, system variables, processes, etc.
USE	Changes the active database.

Replication (Master)	
PURGE MASTER LOGS	Deletes old logging files.
RESET MASTER	Deletes all logging files.
SET SQL_LOG_BIN=0/1	Deactivates/activates binary logging.
SHOW BINLOG EVENTS	Returns a list of all entries in the active logging file (since MySQL 4.0).
SHOW MASTER LOGS	Returns a list of all logging files.
SHOW MASTER STATUS	Specifies the currently active logging file.
SHOW SLAVE HOSTS	Returns a list of all registered slaves (since MySQL 4.0).

Replication (Slave)	
CHANGE MASTER TO	Changes replication settings in `master.info`.
LOAD DATA FROM	Copies all tables from master to slave (since MySQL 4.0).
LOAD TABLE FROM	Copies a table from master to slave.
RESET SLAVE	Reinitializes `master.info`.
SH*OW SLAVE STATUS*	Displays content of `master.info`.
SLAVE START/STOP	Starts and stops replication.

Command Reference (Alphabetical)

In the following reference section, the following syntax is in force:

[option]: Optional parts of a command are shown in square brackets.

variant1 | *variant2* | *variant3*: Alternatives are separated by the | character.

```
ALTER DATABASE [dbname] actions
```

Since MySQL 4.1, with *ALTER DATABASE* you can change global database attributes. The settings are stored in the file `dbname/db.opt`. Instead of *ALTER DATABASE* you can use the equivalent command *ALTER SCHEMA*. If *dbname* is missing, the command applies to the current database.

actions: Currently, two *actions* commands have been implemented.

[DEFAULT] CHARACTER SET charset specifies which character set the database should use by default. (In the definition of tables and columns a different character set can be specified.)

[DEFAULT] COLLATE collname specifies the default sort order.

```
ALTER FUNCTION/PROCEDURE name options
```

ALTER FUNCTION/PROCEDURE since MySQL 5.0 changes details of a stored procedure (SP). However, the command is incapable of changing the code of an SP; for that, you need to delete the SP (*DROP FUNCTION/PROCEDURE*) and then re-create it (*CREATE FUNCTION/PROCEDURE*).

options: *NAME newname* renames the stored procedure.

SQL SECURITY DEFINER/INVOKER changes the security mode of the SP (see *CREATE FUNCTION*).

COMMENT 'newcomment' changes the comment stored with the SP.

```
ALTER TABLE tblname tbloptions
```

ALTER TABLE can be used to change various details of the structure of a table. In the following, we present an overview of the syntactic variants.

In the syntactically simplest form that we shall show here, *ALTER TABLE* changes the table options. The possible options are described in *CREATE TABLE*. The command can be used, for example, to change the type of a table (e.g., from MyISAM to InnoDB).

Note that with many *ALTER TABLE* variants, the table must be re-created. To do this, MySQL creates a new table *X* with the new table properties, and then copies all the records into this new table. Then the existing table is renamed *Y*, and table *X* is renamed *tblname*. Finally, *Y* is deleted. On large tables, this can take considerable time and temporarily use a great deal of hard-disk space.

```
ALTER TABLE tblname ADD newcolname coltype coloptions [FIRST | AFTER existingcolumn]
```

This command adds a new column to a table. The definition of the new column takes place as with *CREATE TABLE*. If the position of the new column is not specified with *FIRST* or *AFTER*, then the new column will be the last column of the table.

The following example adds a new column *ts* with data type *TIMESTAMP* to the *authors* table:

```
ALTER TABLE authors ADD ts TIMESTAMP
```

```
ALTER TABLE tblname ADD INDEX [indexname] (indexcols ...)
ALTER TABLE tblname ADD FULLTEXT [indexname] (indexcols ...)
ALTER [IGNORE] TABLE tblname ADD UNIQUE [indexname] (indexcols ...)
ALTER [IGNORE] TABLE tblname ADD PRIMARY KEY (indexcols ...)
ALTER TABLE tblname ADD SPATIAL INDEX (indexcol)
```

These commands create a new index for a table. If no *indexname* is specified, then MySQL simply uses the name of the indexed column.

The optional key word *IGNORE* comes into play if several identical fields are discovered in the creation of a *UNIQUE* or primary index. Without *IGNORE*, the command will be terminated with an error, and the index will not be generated. With *IGNORE*, such duplicate records are simply deleted.

A spatial index for geometric data can be created starting with MySQL 4.1. The column *indexcol* must have data type *GEOMETRY* and the attribute *NOT NULL*.

```
ALTER TABLE tblname ADD [CONSTRAINT [fr_keyname]]
   FOREIGN KEY [c1_keyname]
   (column1) REFERENCES table2 (column2)
   [ON DELETE {CASCADE | SET NULL | NO ACTION | RESTRICT}]
   [ON UPDATE {CASCADE | SET NULL | NO ACTION | RESTRICT}]
```

This command defines a foreign key constraint. This means that the foreign key *tblname.column1* refers to *table2.column2*, and the table driver should ensure that no references point to nowhere.

This command results in two new indexes being created: the foreign key index for linking *column1* and *column2*, and, if it doesn't exist already, an ordinary index for *tablename.column1*.

Optionally, you can give these indexes names (*ci_keyname* and *fr_keyname*). If you are using a replication system, you should definitely do this; otherwise, it could happen that MySQL uses different names for the original and replicating databases. That can lead later to problems if you wish to delete the foreign key rules.

The optional *ON DELETE* and *ON UPDATE* clauses specify how the table driver is to react to damage to integrity on *DELETE* and *UPDATE* commands (see Chapter 8 for details). By default, the condition is *STRICT*, meaning that potential damage to integrity results in the command not being issued and an error message being triggered.

Currently (MySQL 5.0.*n*), foreign key constraints can be applied only to InnoDB tables. *column2* must be given an index and must be of the same data type as *column1*.

```
ALTER TABLE tblname ALTER colname SET DEFAULT value
ALTER TABLE tblname ALTER colname DROP DEFAULT
```

This command changes the default value for a column or table or deletes an existing default value.

```
ALTER TABLE tblname CHANGE oldcolname newcolname coltype coloptions
```

This command changes the default value for a column in a table or deletes an existing default value. The description of the column proceeds as with *CREATE TABLE*, which you may refer to. If the column name is to remain unchanged, then it must be given twice (that is, *oldcolname* and *newcolname* are identical). Even if *ALTER TABLE* is used only to change the name of a column, both *coltype* and *coloptions* must be completely specified.

```
ALTER TABLE tblname CONVERT TO
    CHARACTER SET  charset [COLLATE collname]
```

This command changes the character set and the optional sort order of all text columns of a table. This change affects not only the formal definition of the table, but its content as well: all text fields of the records are converted.

If you want to change only the definition of the table, and not its content, then you must change the affected column to a BLOB and then back into the desired text data type with the associated character set and sort order. This results in no change to the data, since MySQL leaves BLOB data untouched:

```
ALTER TABLE tblname CHANGE colname colname BLOB
ALTER TABLE tblname CHANGE colname colname VARCHAR(100) CHARACTER SET ...
```

If the text column is equipped with an index, you must first delete the index before the first *ALTER TABLE* command and then re-create it after the second *ALTER TABLE* command.

```
ALTER TABLE tblname DISABLE KEYS
ALTER TABLE tblname ENABLE KEYS
```

Since MySQL 4.0, *ALTER TABLE … DISABLE KEYS* has the effect that all *nonunique* indexes are no longer automatically updated with *INSERT, UPDATE*, and *DELETE* commands. *ALTER TABLE … ENABLE KEYS* restores activation and updating of indexes.

The two commands should be used for carrying out extensive revisions to tables in the most efficient manner possible. (The reconstruction of indexes with *ENABLE KEYS* costs considerably less time than the constant updating with each altered record.)

```
ALTER TABLE dbname.tblname DISCARD TABLESPACE
ALTER TABLE dbname.tblname IMPORT TABLESPACE
```

These two commands are suitable only for InnoDB tables whose data are located in their own files (MySQL server option `innodb_file_per_table`). In MySQL 5.0 it is not permitted to copy such files from one database directory into another or from one MySQL installation to another. The two *ALTER TABLE* variants enable, under certain circumstances, a *tablespace* file to be deactivated and later reactivated.

ALTER TABLE dbname.tblname DISCARD TABLESPACE deletes the table *tblname* and the underlying file `dbname/tblname.ibd`. The file `tblname.frm` is preserved. *ALTER TABLE dbname.tblname*

IMPORT TABLESPACE reactivates the file `dbname/tblname.ibd`. The file must be the result of a backup of the running MySQL installation that was carried out before the *DISCARD TABLESPACE* command.

Further details on these commands, for which there is hardly any useful practical application, can be found at `http://dev.mysql.com/doc/mysql/en/multiple-tablespaces.html`.

```
ALTER TABLE tblname DROP colname
ALTER TABLE tblname DROP INDEX indexname
ALTER TABLE tblname DROP PRIMARY KEY
ALTER TABLE tblname DROP FOREIGN KEY foreign_key_name
```

The first three commands delete a column, an index, or the primary index. The fourth command, since MySQL 4.0.13, deletes the specified foreign key constraint. You can determine with *SHOW CREATE TABLE* the *foreign_key_name* of the index to be deleted.

If you are using replication, you should avoid deleting *FOREIGN KEY* rules. The reason is that MySQL creates a special index when a *FOREIGN KEY* rule is defined. If you do not name this index explicitly, it can happen that different names are used for the original and replicated database.

```
ALTER TABLE tblname ENGINE tabletype
```

This command changes the type of the table (the table driver). Allowed table types include InnoDB and MyISAM. Note that a type change is possible only if the new table driver supports all the properties of the table. For example, the InnoDB table driver supports at present no full-text indexes. If you wish to transform a MyISAM table into an InnoDB table, you must first delete the full-text index (*ALTER TABLE tblname DROP indexname*).

```
ALTER TABLE tblname MODIFY colname coltype coloptions
```

This command functions like *ALTER TABLE … CHANGE* (see above). The only difference is that the column cannot be changed, and thus the name needs to be given only once.

```
ALTER TABLE tblname ORDER BY colname
```

This command re-creates the table and orders the data records by *colname*. If you frequently read records from the table ordered *colname*, this can increase efficiency a bit. The command has no influence over new or changed records, and is therefore useful only if few future changes to the table are expected.

```
ALTER TABLE tblname RENAME AS newtblname
```

This command renames the table (see also *RENAME TABLE*).

```
ALTER TABLE tblname TYPE tabletype
```

This command corresponds to *ALTER TABLE tblname ENGINE tabletype*.

```
ALTER [algoption] VIEW viewname [(columns)] AS command [chkoption]
```

ALTER VIEW changes the properties of a view. It has the same syntax as *CREATE VIEW*. *ALTER VIEW* offers no way to change the name of a view.

```
ANALYZE TABLE tablename1, tablename2, …
```

ANALYZE TABLE performs an analysis of the indexed values of a column. The results are stored, and in the future, this speeds up index access to data records a bit.

With MyISAM tables, the external program `myisamchk -a tblfile` can be used.

```
BACKUP TABLE tblname TO '/backup/directory'
```

BACKUP TABLE copies the files for the specified MyISAM table into a backup directory. The table can be re-created with *RESTORE TABLE*.

Under Unix/Linux, the backup directory for the account under which MySQL is executed must be writable.

BACKUP and *RESTORE* do not work for InnoDB tables. Both commands are considered *deprecated* and are best not used. Alternatives are external backup tools such as `mysqldump`, `mysqlhotcopy` or the InnoDB backup utility.

```
BEGIN
```

If you are working with transaction-capable tables, then *BEGIN* introduces a new transaction. The following SQL commands can then be confirmed with *COMMIT* or revoked with *ROLLBACK*. (All changes to tables are executed only via *COMMIT*.) Further information and examples on the topic of transactions can be found in Chapter 10.

Since MySQL 4.0.11, you can use the ANSI-conforming command *START TRANSACTION* instead of *BEGIN*.

```
CACHE INDEX indexspec1, indexspec2 … IN cachename
```

Since MySQL 4.1, *CACHE INDEX* determines in which cache area a MyISAM index is placed. The command can be used effectively only if first an additional cache area is created (see also the information on structured variables in this chapter).

indexspec: Specifies which MyISAM indexes are to have their caches changed. The following syntax is used for the index specification:

tablename [[INDEX|KEY] (indexname1, indexname2 …)]

If no index name is given, the command holds for all indexes in *tablename*.

cachename: denotes the cache instance in which the indexes are to be placed. Such an area must first be set up with *SET @@cachename.key_buffer_size=n* (*n* is the reserved area size in bytes).

```
CALL spname [parameter1, parameter2 …]
```

This command calls a stored procedure. *CALL* is designed only for user-defined procedures, not for user-defined functions, which must be called with ordinary SQL commands (e.g., with *SELECT*).

```
CHANGE MASTER TO variable1=value1, variable2=value2, …
```

With this command, the replication settings for slave are carried out. The settings are stored in the file `master.info`. The command can be used only for slave computers in a replication system, and it requires the *Super* privilege. It recognizes the following variable names:

MASTER_HOST specifies the hostname or IP number of the master computer.

MASTER_USER specifies the user name used for communication with the master computer.

MASTER_PASSWORD specifies the associated password.

MASTER_PORT specifies the port number of the master computer (normally 3306).

MASTER_LOG_FILE specifies the current logging file on the master computer.

MASTER_LOG_POS specifies the current read position within the logging file on the master computer.

```
CHECK TABLE tablename1, tablename2 … [ TYPE=QUICK ]
```

CHECK TABLE tests the internal integrity of the database file for the specified table. Any errors that are discovered are not corrected. With MyISAM tables, instead of this command, the external program `myisamchk -m tblfile` can be used.

```
COMMIT
```

COMMIT ends a transaction and stores all changes in the database. (Instead of executing *COMMIT*, you can cancel the pending changes with *ROLLBACK*.) *BEGIN/COMMIT/ROLLBACK* function only if you are working with transaction-capable tables. Further information and examples on transactions can be found in Chapter 10.

```
CREATE  DATABASE  [IF NOT EXISTS] dbname [options]
```

CREATE DATABASE generates the specified database. (More precisely, an empty directory is created in which tables belonging to the new database can be stored.) Note that database names are case sensitive. This command can be executed only if the user has sufficient access privileges to create new databases. Instead of *CREATE DATABASE* the equivalent command *CREATE SCHEMA* can be used.

options: Since MySQL 4.1, you can specify the default character set for a table:

[DEFAULT] CHARACTER SET charset [COLLATE collname].

The optional key word *DEFAULT* has no function (that is, it does not matter whether you specify it or not). With *COLLATE* you can select the sort order if there is more than one for the character set in question. If you do not specify the *CHARACTER SET*, then the default character set of the server is used.

CREATE FUNCTION name ([parameters]) RETURNS datatype [options] code

Since MySQL 5.0, *CREATE FUNCTION* creates a user-defined function (stored procedure) in the current database. Defining stored procedures requires the *Super* privilege and executing them requires the *Execute* privilege.

name: Specifies the function name. It is allowed for a function and a procedure within a single database to have the same name (see *CREATE PROCEDURE*).

parameters: Several parameters may be specified, separated by commas. Each parameter must be followed by its data type, e.g., *para1 INT*, *para2 BIGINT*.

datatype: Gives the data type of the function's return value. All MySQL data types are allowed, e.g., *INT*, *DOUBLE*, *VARCHAR(n)*.

options: The following options can be used in function definition:

LANGUAGE SQL: Specifies the language of the stored procedure code. The only permissible *LANGUAGE* setting is currently SQL. This setting holds by default. Future versions of MySQL will offer the option of defining SPs in other programming languages (e.g., PHP).

[NOT] DETERMINISTIC: An SP is considered deterministic if it always returns the same results with the same parameters. (SPs whose result depends on the content of the database are not deterministic.) By default, SPs are nondeterministic. Deterministic SPs can be executed particularly efficiently. (For example, it is possible to store the result for particular parameters in a cache.) Currently, however, the *DETERMINISTIC* option is ignored by the MySQL optimization functions.

SQL SECURITY DEFINER/INVOKER: The *SECURITY* mode specifies the access privileges under which the SP should be executed. SPs that are defined with *SQL SECURITY DEFINER* have the same privileges as the MySQL user who defined the SP. This is the default security mode. SPs defined with the option *SQL SECURITY INVOKER* have the access privileges of the MySQL user who executed the SP.

COMMENT 'text': The comment text is stored together with the SP.

code: The actual SP code is usually given in the form of an SQL command. If the SP consists of more than one command, they must be placed between *BEGIN* and *END* and separated by semicolons (details are in Chapter 13).

Here is an example:

```
CREATE FUNCTION half(a INT) RETURNS INT
BEGIN
  RETURN a/2;
END
```

CREATE FUNCTION name RETURNS datatype SONAME libraryname

CREATE FUNCTION makes possible not only the definition of SPs, but also the binding of a function to an external library into MySQL. Such functions are called *user-defined functions*, or UDFs for short. Programming such functions in C or C++ requires a degree of background knowledge about how functions work in MySQL (and, of course, the requisite tools, like compilers). Further information can be found at `http://dev.mysql.com/doc/mysql/en/extending-mysql.html` and `http://mysql-udf.sourceforge.net/`.

```
CREATE [UNIQUE|FULLTEXT] INDEX indexname ON tablename (indexcols …)
```

CREATE INDEX enlarges an existing database to include an index. As *indexname*, the name of the column is generally used. *CREATE INDEX* is not a freestanding command, but merely an alternative form of *ALTER TABLE ADD INDEX/UNIQUE*, which you should see for details.

```
CREATE PROCEDURE name ([parameters]) [options] code
```

CREATE PROCEDURE creates, since MySQL 5.0, a user-defined procedure (stored procedure) in the current database. The definition of stored procedures requires the *Super* privilege, while execution requires the *Execute* privilege.

The syntax of this command is very similar to that of *CREATE FUNCTION*. However, unlike a function, a procedure cannot return a result directly. However, a *SELECT* command can be executed and parameters changed. For this reason, the syntax for the parameter list is a bit different from that of *CREATE FUNCTION*.

parameters: More than one parameter can be defined, and they are separated by commas. Each parameter is defined as follows:

[IN or OUT or INOUT] parametername datatype

The key words *IN*, *OUT*, and *INOUT* specify whether the parameter is only for input, only for output, or for data transport in both directions (default is *IN*). All MySQL data types are allowed, such as *INT*, *VARCHAR(n)*, *DOUBLE*.

Here is an example:

```
CREATE PROCEDURE half(IN a INT, OUT b INT)
BEGIN
  SET b=a/2;
END
```

```
CREATE  [TEMPORARY]  TABLE  [IF NOT EXISTS]  tblname
  (colname1 coltype coloptions reference,
    colname2 coltype coloptions reference...
[ , index1, index2 ...]
  )
  [tbloptions]
```

CREATE TABLE generates a new table in the current database. If a database other than the current one is to be used, the table name can be specified in the form *dbname.tblname*. If the table already exists, an error message results. There is no error message if *IF NOT EXISTS* is used, but in this case, the existing table is not affected, and no new table is created.

With the key word *TEMPORARY* a temporary table is created. If a temporary table is created and a like-named, but not temporary, table already exists, the temporary table is created without an error message. The old table is preserved, but it is masked by the temporary table. If you want your temporary table to exist only in RAM (for increased speed), you must also specify *ENGINE = HEAP*.

The creation of regular tables requires the *Create* privilege. Since version 4.0, the creation of temporary tables requires the *Create Temporary Table* privilege.

colname: Name of the column.

coltype: Data type of the column. A list of all MySQL data types (*INT*, *TEXT*, etc.) appears earlier in this chapter.

coloptions: Here certain attributes (options) can be specified:

NOT NULL | NULL

UNSIGNED

ZEROFILL

BINARY

DEFAULT defaultval | DEFAULT CURRENT_TIMESTAMP

ON UPDATE CURRENT_TIMESTAMP

AUTO_INCREMENT | IDENTITY

PRIMARY KEY

CHARACTER SET charset [COLLATE collname]

COMMENT text

reference: MySQL provides various key words for the declaration of foreign keys in keeping track of referential integrity, e.g., *REFERENCES tblname (idcolumn)*. These key words are currently ignored, however (and with no error message). Even if you use InnoDB tables, the foreign key constraints for such tables must be specified within the confines of the index definition.

index: *KEY* or *INDEX* defines a usual index spanning one or more columns. *UNIQUE* defines a unique index (that is, in the column or columns, no identical values or groups of values can be stored). With both variants an arbitrary index name may be given for the internal management of the index. *PRIMARY KEY* likewise defines a *UNIQUE* index. Here, however, the index name is predefined: It is, not surprisingly, *PRIMARY*. With *FULLTEXT*, an index is declared for full-text search (in MySQL 4.0, only with MyISAM tables). Since MySQL 4.1, an index for *GEOMETRY* data can be created with *SPATIAL INDEX*; *indexcol* must be defined with the attribute *NOT NULL*.

KEY | INDEX [indexname] (indexcols ...)

UNIQUE [INDEX] [indexname] (indexcols ...)

PRIMARY KEY (indexcols ...)

FULLTEXT [indexname] (indexcols ...)

SPATIAL INDEX [indexname] (indexcol)

Foreign key constraints: If you are using InnoDB tables, here you can formulate foreign key constraints. The syntax is as follows:

[CONSTRAINT [fr_keyname]]

FOREIGN KEY [c1_keyname] (column1) REFERENCES table2 (column2)

[ON DELETE {CASCADE | SET NULL | NO ACTION | RESTRICT}]

[ON UPDATE {CASCADE | SET NULL | NO ACTION | RESTRICT}]

This means that *tblname.column1* is a foreign key that refers to *table2.column2*. More details can be found under *ALTER TABLE … ADD FOREIGN KEY*.

tbloptions: Here various table options can be specified, though here we shall exhibit only the most important of them. Not all options are possible with every table type. Information on the different table types and their variants can be found in Chapter 8.

ENGINE = MYISAM | HEAP | INNODB

ROW_FORMAT= default | dynamic | static | compressed

AUTO_INCREMENT gives the initial value for the counter of an *AUTO_INCREMENT* column (e.g., 100000 if you wish to have six-digit integers). *CHECK_SUM=1* has the effect that a check sum is stored for each data record, which helps in reconstruction if the database is damaged. *PACK_KEYS=1* results in a smaller index file. This speeds up read access, but slows down changes. *DELAY_KEY_WRITE = 1* results in indexes not being updated each time a change to a record is made. Rather, they are updated every now and then.

AUTO_INCREMENT = n

CHECKSUM = 0 | 1

PACK_KEYS = 0 | 1

DELAY_KEY_WRITE = 0 | 1

With *COMMENT* you can save a brief text, for example, to describe the purpose of the table. The comment can be read with *SHOW CREATE DATABASE dbname*:

COMMENT= 'comment'.

Since MySQL 4.1, you can specify the character set and sort order for a table. (You can also set these parameters for a single column.) If no character set is specified, then the default character set for the table or that of the MySQL server is used:

[DEFAULT] CHARACTER SET charset [COLLATE collname].

The *CREATE TABLE* syntax contains some duplication. For example, a primary index can be declared in two different ways, either as an attribute of a column (*coloptions*) or as an independent index (*index*). The result is, of course, the same. It is up to you to decide which form you prefer.

MySQL has the property that in many cases changes the definition of a column, for example by providing a suitable *DEFAULT* value if none is specified (*silent column changes*; see also Chapter 9). For this reason it is recommended that after you create a table, you take a look at the actual MySQL table definition with *SHOW CREATE TABLE name*.

Here is an example:

```
CREATE TABLE test (id     INT NOT NULL AUTO_INCREMENT,
                   data   INT NOT NULL,
                   txt    VARCHAR(60),
                   PRIMARY KEY (id))
```

```
CREATE  [TEMPORARY]  TABLE  [IF NOT EXISTS]  tblname
  [(newcolname1 coltype coloptions reference,
    newcolname2 coltype coloptions reference …
    [ , key1, key2 …]
  )]
  [tbloptions]
  [IGNORE | REPLACE] SELECT …
```

With this variant of the *CREATE TABLE* command, a table is filled with the result of a *SELECT* command. The individual columns of the new table take their types from the data types of the *SELECT* command and thus do not have to be (and may not be!) declared explicitly.

Unfortunately, neither indexes nor attributes such as *AUTO_INCREMENT* are carried over from the old table. There can also be changes in column types, such as *VARCHAR* columns turning into *CHAR*.

If an index is to be created in the new table for individual columns (e.g., *PRIMARY KEY (id)*), then this can be specified. Moreover, there is the option of defining new columns (e.g., an *AUTO INCREMENT* column.

The key words *IGNORE* and *REPLACE* specify how MySQL should behave if several records with the same value are placed by the command into a *UNIQUE* column. With *IGNORE*, the existing record is retained, and new records are ignored. With *REPLACE* existing records are replaced by the new ones. If neither option is used, an error message results.

If a table is to be copied one to one, it is better to create the new table with *CREATE TABLE table2 LIKE table1* (since MySQL 4.1) and then copy in the data with *INSERT INTO table2 SELECT * FROM table1*.

Here is an example:

```
CREATE TEMPORARY TABLE tmp
  SELECT id, authName FROM authors WHERE id<20
```

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] newtable LIKE oldtable
```

This command, since MySQL 4.1, creates a new, empty table *newtable* corresponding to the declaration of the existing table *oldtable*.

```
CREATE TRIGGER name time event
  ON tablename
  FOR EACH ROW code
```

CREATE TRIGGER defines SQL code that in the future will be executed automatically before or after particular database commands for the affected table. For each event only one trigger procedure is allowed. The execution of *CREATE TRIGGER* requires the *Super* privilege.

> *name*: Specifies the name of the trigger. Currently (MySQL 5.0.3) like-named triggers are allowed within a single database if they are defined for different tables. In future MySQL versions a trigger name must be unique for the entire database.
>
> *time*: *BEFORE | AFTER*
>
> Specifies whether the trigger code is to be executed before or after the trigger event.
>
> *event*: *INSERT | UPDATE | DELETE*
>
> Specifies for what database operations the trigger code is to be executed.
>
> *tablename*: Specifies for which table the trigger is defined.
>
> *code*: Gives the trigger code. The syntax is the same as for a stored procedure.

Here is an example:

```
CREATE TRIGGER test_before_insert
  BEFORE INSERT ON test FOR EACH ROW
BEGIN
  IF NEW.percent < 0.0 OR NEW.percent > 1.0 THEN
    SET NEW.percent = NULL;
  END IF;
END
```

```
CREATE [algoption] VIEW viewname [(columns)] AS command [chkoption]
```

CREATE VIEW creates, since MySQL 5.0, a *View*. This is a virtual table based on a *SELECT* command. If you wish to replace an existing *View*, execute the command in the form *CREATE OR REPLACE ...*. The command requires the *Create View* privilege.

algoption: *ALGORITHM = UNDEFINED | MERGE | TEMPTABLE*

ALGORITHM tells how the *View* is to be represented internally. This option was not documented as this book was being completed. By default, MySQL always uses *UNDEFINED* (can be determined with *SHOW CREATE TABLE viewname*).

viewname: Specifies the name of the *View*. The same rules hold as for table names. The name of the *View* may not be the same as that of the table.

columns: The columns of a *View* have as a rule the same names and data types as the columns of the underlying table. With *columns* you can specify different names and data types. The syntax is the same as for *CREATE TABLE*.

command: Here you give a *SELECT* command. The data in the *View* are the result of this command.

chkoption: *WITH [CASCADED | LOCAL] CHECK OPTION*

WITH CHECK OPTION means that changes to the *View* records are allowed only if the *WHERE* conditions of the *SELECT* command are satisfied. *WITH CHECK OPTION* is of course relevant only if the *View* is changeable.

The variant *WITH LOCAL CHECK OPTION* affects *Views* that have been derived from other *Views*. *LOCAL* means that only the *WHERE* conditions of the *CREATE VIEW* command are considered, and not the *WHERE* conditions of higher-level *Views*.

The opposite effect is achieved with *WITH CASCADED CHECK OPTION*: Now the *WHERE* conditions of all higher-level *Views* are considered. If you specify neither *CASCADED* nor *LOCAL*, the default is *CASCADED*.

Here is an example:

```
CREATE VIEW v1 AS
  SELECT titleID, title, subtitle FROM titles
  ORDER BY title, subtitle
```

```
DELETE [deleteoptions] FROM tablename
  [WHERE condition]
  [ORDER BY ordercolumn [DESC]]
  [ LIMIT maxrecords ]
```

DELETE deletes the records in a table encompassed by *condition*.

deleteoptions: *LOW_PRIORITY, QUICK, IGNORE*

LOW_PRIORITY has the effect that the data records are deleted only when all read operations are complete. (The goal of this option is to avoid having *SELECT* queries unnecessarily delayed due to *DELETE* operations.)

QUICK has the effect that during deletion, an existing index is not optimized. This speeds up the *DELETE* command, but it can lead to a somewhat inefficient index.

IGNORE has the effect since MySQL 4.1 that the *DELETE* command is continued even if errors occur. All errors are converted to warnings, which can be read with *SHOW WARNINGS*.

condition: This condition specifies which records are to be deleted. For the syntax of *condition*, see *SELECT*.

ordercolumn: With *ORDER BY* you can first sort the data to be deleted. This makes sense only in combination with *LIMIT*, in order, for example, to delete the first or last ten records (according to some sort criterion).

maxrecords: With *LIMIT*, the maximum number of records that may be deleted is specified.

If *DELETE* is executed without conditions, then all records of the table are deleted (so be careful!). *DELETE* without conditions cannot be part of a transaction. If a transaction is open, it is closed with *COMMIT* before the *DELETE* command is executed. If you wish to delete large tables completely, it is more efficient to use the command *TRUNCATE*.

```
DELETE [deleteoptions]  table1, table2 … FROM table1, table2, table3 …
  [USING columns]
  WHERE conditions
```

This variant of *DELETE* (available since version 4.0) deletes records from tables *table1*, *table2*, etc., where the data of additional tables (*table3*, etc.) are considered in the search criteria.

After *DELETE*, all tables from which data are to be deleted must be specified. After *FROM*, all *DELETE* tables must appear, as well as any additional tables that serve only in formulating the search criteria.

deleteoptions: Here you can specify options as in a usual *DELETE* command.

columns: Here fields that link the tables can be specified (see also *SELECT*). This assumes that the linking field has the same name in both tables.

conditions: In addition to the usual delete criteria, here one may specify linking conditions (e.g., *WHERE table1.id = table2.forgeinID*).

```
DESCRIBE tablename [ columnname ]
```

DESCRIBE returns information about the specified table in the current database (or about a particular column of this table). Instead of *columnname*, a pattern with the wild cards _ and % can be given. In this case, *DESCRIBE* displays information about those columns matching the pattern. *DESCRIBE* returns the same information as *EXPLAIN* or *SHOW TABLE* or *SHOW COLUMN*.

```
DO selectcommand
```

DO is a variant of *SELECT* and has basically the same syntax. The difference between the two is that *DO* returns no results. For example, *DO* can be used for variable assignment, for which it is somewhat faster than *SELECT* (thus, for example, *DO @var:=3*).

```
DROP DATABASE [IF EXISTS] dbname
```

DROP DATABASE deletes an existing database with all of its data. This cannot be undone, so be careful! If the database does not exist, then an error is reported. This error can be avoided with an *IF EXISTS*.

In the execution of this command, all files in the directory `dbname` with the following endings are deleted, among others: `.BAK`, `.DAT`, `.HSH`, `.ISD`, `.ISM`, `.MRG`, `.MYD`, `.MYI`, `.db`, `.frm`, as well as the file `db.opt`.

```
DROP FUNCTION fnname
```

DROP FUNCTION deletes the specified stored procedure or deactivates an external auxiliary function that was made available to MySQL earlier with *CREATE FUNCTION*.

```
DROP INDEX indexname ON tablename
```

DROP INDEX removes an index from the specified table. Usually, *indexname* is the name of the indexed column, or else *PRIMARY* for the primary index.

```
DROP PROCEDURE  prname
```

DROP PROCEDURE deletes the specified stored procedure.

```
DROP [TEMPORARY] TABLE [IF EXISTS] tablename1, tablename2 … [options]
```

DROP TABLE deletes the specified (temporary) tables irrevocably. The option *IF EXISTS* avoids an error message if the tables do not exist.

Note that *DROP TABLE* automatically ends a running transaction (*COMMIT*).

DROP TEMPORARY TABLE (available since MySQL 4.1) deletes only temporary tables. In contrast to the ordinary *DROP TABLE* command, this one has no influence over running transactions.

options: *RESTRICT | CASCADE*

The two options *RESTRICT* und *CASCADE* are currently nonfunctional. They currently should simplify the porting of SQL code to other database systems.

```
DROP TRIGGER tablename.triggername
```

DROP TRIGGER deletes the specified trigger.

```
DROP VIEW [IF EXISTS] viewname1, viewname2 … [options]
```

DROP VIEW deletes the specified *Views* irrevocably. The same options are available as for *DROP TABLE*.

```
EXPLAIN tablename
```

EXPLAIN returns a table with information about all the columns of a table (field name, field type, index, default value, etc.). The same information can be determined as well with *SHOW COLUMNS* or *DESCRIBE*, or via an external program such as `mysqlshow`.

```
EXPLAIN SELECT selectcommand
```

EXPLAIN SELECT returns a table with information about how the specified *SELECT* command was executed. These data can help in speed optimization of queries, and in particular in deciding which columns of a table should be indexed. (The syntax of *selectcommand* was described under *SELECT*. An example of the use of *EXPLAIN SELECT* and a brief description of the resulting table can be found in Chapter 8.)

```
FLUSH flushoptions
```

FLUSH empties the MySQL internal intermediate storage. Any information not stored already is thereby stored in the database. The execution of *FLUSH* requires the *RELOAD* privilege.

flushoptions: Here one may specify which cache(s) should be emptied. Multiple options should be separated by commas.

DES_KEY_FILE: Reloads the key files for the functions *DES_ENCRYPT* and *DES_DECRYPT*.

HOSTS: mpties the host cache table. This is necessary especially if in the local network the arrangement of IP numbers has changed.

LOGS: Closes all logging files and then reopens them. In the case of update logs, a new logging file is created, whereby the number of the file ending is increased by 1 (*name.000003*➤ *name.00004*). With error logs the existing file is renamed to `name.old` and a new error log file is created.

QUERY CACHE: Defragments the query cache so that it can use its memory more efficiently. The cache is not emptied. (If you wish to do this, execute *RESET QUERY CACHE*.)

PRIVILEGES: Reloads the privileges database *mysql* (corresponds to `mysqladmin reload`).

STATUS: Sets most status variables to 0.

TABLES: Closes all open tables.

TABLE[S] tblname1, tblname2, ...: Closes the specified tables.

TABLES WITH READ LOCK: As above, except that additionally, *LOCK* is executed for all tables, which remains in force until the advent of a corresponding *UNLOCK table*.

USER_RESOURCES: Resets the counters for *MAX_QUERIES_PER_HOUR*, *MAX_UPDATES_PER_HOUR*, and *MAX_CONNECTIONS_PER_HOUR* (see *maxlimits* under *GRANT*).

Most *FLUSH* operations can also be executed through the auxiliary program *`mysqladmin`*.

```
GRANT privileges ON objects
  TO users [IDENTIFIED BY 'password']
  [REQUIRE ssloptions ]
  [WITH GRANT OPTION | maxlimits ]
```

GRANT helps in the allocation of access privileges to database objects.

ALTER, CREATE, CREATE TEMPORARY TABLES, CREATE VIEW, DELETE, DROP, EXECUTE, FILE, INDEX, LOCK TABLE, PROCESS, REFERENCES, RELOAD, REPLICATION CLIENT, REPLICATION SLAVE, SELECT, SHOW DATABASE, SHOW VIEW, SHUTDOWN, SUPER, UPDATE

If you wish to set all (or no) privileges, then specify *ALL* (or *USAGE*). (The second variant is useful if you wish to create a new MySQL user to whom as of yet no privileges have been granted.) The *Grant* privilege can be set only via *WITH GRANT OPTION*; that is, *ALL* does not include the *Grant* privilege.

If the privileges are to hold only for certain columns of a table, then specify the columns in parentheses. For example, you may specify *GRANT SELECT(columnA, columnB)*.

objects: Here databases and tables are specified. The following syntactic variants are available:

databasename.tablename	only this table in this database
databasename.spname	only this stored procedure
*databasename.**	all tables in this database
tablename	only this table in the current database
*	all tables of the current database
.	global privileges

Wild cards may not be used in the database names.

users: Here one or more (comma-separated) users may be specified. If these users are not yet known to the *user* table, they are created. The following variants are allowed:

username@hostname	only this user at *hostname*
'username'@'hostname'	as above, with special characters
username	this user at all computers
''@hostname	all users at *hostname*
''	all users at all computers

password: Optionally, with *IDENTIFIED BY*, a password in plain text can be specified. *GRANT* encrypts this password with the function *PASSWORD* before it is entered in the *user* table. If more than one user is specified, then more than one password may be given:

TO user1 IDENTIFIED BY 'pw1', user2 IDENTIFIED BY 'pw2', ...

ssloptions: If access to MySQL is to be SSL encrypted or if user identification is to take place with X509, you can specify the required information for establishing the connection here. The syntax is as follows:

REQUIRE SSL | X509 [ISSUER 'iss'] [SUBJECT 'subj'] [CIPHER 'ciph']

REQUIRE SSL means that the connection must be SSL encrypted (thus a normal connection is not permitted). *REQUIRE X509* means that the user must possess a valid certificate for identification that meets the X509 standard.

ISSUER specifies the required issuer of the certificate. (Without *ISSUER*, the origin of the certificate is not considered.)

SUBJECT specifies the required content of the certificate's *subject* field. (Without *SUBJECT*, the content is not considered.)

CIPHER specifies the required SSL encryption algorithm. (SSL supports various algorithms. Without this specification, all algorithms are allowed, including older ones that may have security loopholes.)

maxlimits: Here you can specify how many connections per hour the user is allowed to establish, as well as the number of *SELECT* and *INSERT/UPDATE/DELETE* commands. The default setting for all three values is 0 (no limit):

MAX_QUERIES_PER_HOUR n

MAX_UPDATES_PER_HOUR n

MAX_CONNECTIONS_PER_HOUR n

If the specified user does not yet exist and *GRANT* is executed without *IDENTIFIED BY*, then the new user has no password (which represents a security risk). On the other hand, if the user already exists, then *GRANT* without *IDENTIFIED BY* does not alter the password. (There is thus no danger that a password can be accidentally deleted by *GRANT*.)

It is impossible with *GRANT* to delete privileges that have already been granted (for example, by executing the command again with a smaller list of privileges). If you wish to take away privileges, you must use *REVOKE*.

GRANT may be used only by users with the *Grant* privilege. The user that executes *GRANT* may bestow only those privileges that he himself possesses. If the MySQL server is started with the option `safe-user-create`, then to create a new user, a user needs the *Insert* privilege for the table *mysql.user* in addition to the *Grant* privilege.

Since MySQL 5.0.3 *GRANT* can be used to create new users only by those with the *Create User* privilege.

```
HANDLER tablename OPEN [AS aliasname]
HANDLER tablename READ  FIRST|NEXT   [ WHERE condition  LIMIT n, m ]
HANDLER tablename READ indexname  FIRST|NEXT|PREV|LAST [ WHERE … LIMIT …]
HANDLER tablename CLOSE
```

Since MySQL 4.0.3, *HANDLER* enables direct access to MyISAM and InnoDB tables. This command can be used as a more efficient substitute for simple *SELECT* commands. This is particularly true if records are to be processed one at a time or in small groups.

The command is easy to use: First, access to a table is achieved with *HANDLER OPEN*. Then, *HANDLER READ* may be executed as often as you like, generally the first time with *FIRST*, and thereafter with *NEXT*, until no further results are forthcoming. The command returns results as with *SELECT ** (that is, all columns). *HANDLER CLOSE* terminates access.

HANDLER tablename READ reads the records in the order in which they were stored. On the other hand, the variant *HANDLER tablename READ indexname* uses the specified index. If you wish to use a primary index, you must use the form *`primary`*.

HANDLER was not conceived for use with typical MySQL applications, if for no other reason than that the code would be completely incompatible with every other database server. *HANDLER* is suitable for programming low-level tools (e.g., backup tools or drivers that simulate simple data access with a cursor). Note that *HANDLER* does not block tables (no locking), and therefore, the table can change while its data are being read.

HANDLER should not be used in stored procedures. Instead, use a cursor. *HANDLER* also has nothing to do with the command *DECLARE HANDLER*, which is used for error-handling in stored procedures.

```
HELP
HELP contents
HELP functionname
```

Since MySQL 4.1, *HELP* returns a brief help text. Instead of *HELP* the abbreviation *?* can be used.

```
INSERT [options1]  [INTO]  tablename [(columnlist)]
    VALUES (valuelist1), (…), …   [options2]

INSERT [options1]  [INTO]  tablename
    SET column1=value1, column2=value2 … [options2]

INSERT [options1]  [INTO]  tablename [ (columnlist) ]
    SELECT …
```

The *INSERT* command has the job of inserting new records into an existing table. There are three main syntax variants. In the first (and most frequently used) of these, new data records are specified in parentheses. Thus a typical *INSERT* command looks like this:

```
INSERT INTO tablename (columnA, columnB, columnC)
VALUES ('a', 1, 2), ('b', 7, 5)
```

The result is the insertion of two new records into the table. Columns that are allowed to be *NULL*, for which there is a default value, or which are automatically filled in by MySQL via *AUTO_IN* do not have to be specified. If the column names (i.e., in *columnlist*) are not given, then in *VALUES* all values must be given in the order of the columns.

With the second variant, only one record can be changed (not several simultaneously). Such a command looks like this:

```
INSERT INTO tablename SET columnA='a', columnB=1, columnC=2
```

For the third variant, the data to be inserted come from a *SELECT* instruction.

options1: The behavior of this command can be controlled with options:

IGNORE has the effect that the insertion of records with existing values is simply ignored for *UNIQUE KEY* columns. (Without this option, the result would be an error message.)

LOW_PRIORITY | *DELAYED* | *HIGH_PRIORITY* have influence over when the insertion operation is carried out.

In *LOW_PRIORITY* and *DELAYED*, MySQL delays its storage operation until there are no pending read accesses to the table. The advantage of *DELAYED* is that MySQL returns OK at once, and the client does not need to wait for the end of the saving operation. However, *DELAYED* cannot be used if then an *AUTO_INCREMENT* value with *LAST_INSERT_ID( )* is to be determined. *DELAYED* should also not be used if a *LOCK* was placed on the table. (The reason is this: For executing *INSERT DELAYED*, a new MySQL thread is started, and table locking uses threads in its operation.)

The records to be inserted are stored in RAM until the insertion operation has actually been carried out. If MySQL should be terminated for some reason (crash, power outage), then the data are lost.

HIGH_PRIORITY normally has no effect; that is, *INSERT* inserts the data at once. *HIGH_PRIORITY* is designed only for the case that the server was started with the option `low-priority-updates`. This option has the effect that *INSERT* commands are stored in LOW_PRIORITY mode. *HIGH_PRIORITY* overwrites this default setting.

options2: *ON DUPLICATE KEY UPDATE column1=value1, column2=value2 …*

When a *UNIQUE* or *PRIMARY* index is violated during the insertion of new data, this option has the effect since MySQL 4.1 of replacing the existing record. If *id* is a *PRIMARY* column and an entry with *id=1* already exists, then

INSERT INTO tablename (id, data) VALUES (1, 10)

ON DUPLICATE KEY UPDATE data=data+10

has the same effect as

UPDATE tablename SET data=data+10 WHERE id=1.

Here *VALUE(columnname)* can be used in the column allocation. This function returns the value of the affected column. It is useful when a general *UPDATE* instruction is to be formulated for several records:

```
INSERT INTO tablename (id, data) VALUES (1, 10), (2, 15)
ON DUPLICATE KEY UPDATE data=data+VALUE(data)
```

Note that in using default values, there are special rules for *TIMESTAMP* and *AUTO_INCREMENT* values. (These rules hold for *UPDATE* commands as well.)

Columns with default values: If you want MySQL to use the default value for a column, then either do not specify this column in your *INSERT* command, or pass an empty character string (not *NULL*) as the value:

```
INSERT INTO table (col1, col2_with_default_value) VALUES ('abc', '')
```

***TIMESTAMP* columns:** If you want MySQL to insert the current time in the column, then either omit this column in your *INSERT* command, or pass the value *NULL* (not an empty character string). It is also allowed to pass a character string with a timestamp value if you wish to store a particular value.

***AUTO_INCREMENT* columns:** Here as well, either you don't pass the column, or you pass the value *NULL* if MySQL is to determine the *AUTO_INCREMENT* value itself. You may pass any other value that is not otherwise in use.

```
JOIN
```

JOIN is not actually an SQL command. This key word is mostly used as part of a *SELECT* command, to link data from several tables. *JOIN* will be described under *SELECT*.

```
KILL threadid
```

This command terminates a specified thread (subprocess) of the MySQL server. It is allowed only to those users who possess the *Super* privilege. A list of running threads can be obtained via *SHOW PROCESSLIST* (where again, this command assumes the *Process* privilege). Threads can also be terminated via the external program `mysqladmin`.

```
LOAD DATA  [ loadoptions ]  INFILE 'filename'  [ duplicateopt ]
  INTO TABLE tablename
  [ importopt ]
  [ IGNORE ignorenr LINES ]
  [ (columnlist) ]
```

LOAD DATA reads a text file and inserts the data contained therein line by line into a table as data records. *LOAD DATA* is significantly faster then inserting data by multiple *INSERT* commands.

Normally, the file *filename* is read from the server's file system, on which MySQL is running. (For this, the *FILE* privilege is required. For security reasons, the file must either be located in the directory of the database or be readable by all users of the computer.)

If the text file to be imported has characters outside the ASCII character set, you must set the character set of the text with *SET NAMES* before the command *LOAD DATA*.

loadoptions: *LOCAL* has the effect that the file *filename* on the local client computer is read (that is, the computer on which the command *LOAD DATA* is executed, not on the server computer). For this, no *FILE* privilege is necessary. (The *FILE* privilege relates only to the file system of the MySQL server computer.) Note that *LOAD DATA LOCAL* can be deactivated, depending on how the MySQL server was compiled and configured (option `local-infile`).

LOW PRIORITY has the effect that the data are inserted into the table only if no other user is reading the table.

CONCURRENT makes it possible in many cases for data to be inserted into a table and read out at the same time (by other clients). However, this works only for MyISAM tables and only when the new records are inserted exclusively at the end of the table file. The table file is not allowed to contain any free memory (holes). You can ensure this via *OPTIMIZE TABLE*.

filename: If a file name is given without the path, then MySQL searches for this file in the directory of the current database (e.g., *'bulk.txt'*).

If the file name is given with a relative path, then the path is interpreted by MySQL relative to the `data` directory (e.g., *'mydir/bulk.txt'*).

File names with absolute paths are taken without alteration (for example, *'/tmp/mydir/bulk.txt'*).

duplicateoptions: *IGNORE | REPLACE* determine the behavior of MySQL when a new data record has the same *UNIQUE* or *PRIMARY KEY* value as an existing record. With *IGNORE,* the existing record is preserved, and the new records are ignored. With *REPLACE,* existing records are replaced by the new ones. If neither of these options is used, then the result is an error message.

importoptions: Here is specified how the data should be formatted in the file to be imported. The entire *importoptions* block looks like this:

[FIELDS

[TERMINATED BY 'fieldtermstring']

[ENCLOSED BY 'enclosechar']

[ESCAPED BY 'escchar']]

[LINES TERMINATED BY 'linetermstring']

fieldtermstring specifies the character string that separates the individual columns within the row (e.g., a tab character).

enclosechar specifies the character that should stand before and after individual entries in the text file (usually the single or double quote character for character strings). If an entry begins with this character, then that character is removed from the beginning and end. Entries that do not begin with the *enclosechar* character will still be accepted. The use of the character in the text file is thus to some extent optional.

escchar specifies which character is to be used to mark special characters (usually the backslash). This is necessary if special characters appear in character strings in the text file that are also used to separate columns or rows. Furthermore, MySQL expects the zero-byte in the form \0, where the backslash is to be replaced as necessary by *escchar* if a character has been specified for *escchar*).

linetermstring specifies the character string with which rows are to be terminated. With DOS/Windows text files this must be the character string *' \r\n'*.

In these four character strings, the following special characters can be specified:

\0	0 byte
\b	backspace
\n	newline
\r	carriage return
\s	space
\t	tab
\'	single quote (')
\"	double quote (")
\\	backslash

Furthermore, the character strings can be given in hexadecimal form (e.g., *0x22* instead of '\"').

If no character strings are given, then the following is the default setting:

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

LINES TERMINATED BY '\n'

ignorenr: This value specifies how many lines should be ignored at the beginning of the text file. This is particularly useful if the first lines contain table headings.

columnlist: If the order of the columns in the text file does not exactly correspond to that in the table, then here one may specify which file columns correspond with which table columns. The list of columns must be set in parentheses: for example, *(firstname, lastname, birthdate)*.

If *TIMESTAMP* columns are not considered during importation or if *NULL* is inserted, then MySQL inserts the actual time. MySQL exhibits analogous behavior with *AUTO_INCREMENT* columns.

LOAD DATA displays as result, among other things, an integer representing the number of warnings.

Starting with MySQL 4.1 you will can display all warnings and errors caused by *LOAD DATA* with the commands *SHOW WARNINGS* and *SHOW ERRORS*.

Instead of *LOAD DATA*, you can also use the program `mysqlimport`. This program creates a link to MySQL and then uses *LOAD DATA*. The inverse of *LOAD DATA* is the command *SELECT … INTO OUTFILE*. With it you can export a table into a text file. Further information and concrete examples can be found in Chapter 14.

```
LOAD DATA FROM MASTER
```

This command since MySQL 4.0 copies all MyISAM tables from master to slave of a replication system. The tables of the *mysql* database are not copied. After copying, replication is begun on the slave (that is, the variables *MASTER_LOG_FILE* and *MASTER_LOG_POS* are set, which normally must be set with *CHANGE MASTER TO*).

This command can be used in many cases, in particular when no InnoDB tables are being used, for a convenient setting up of a replication system. It assumes that the replication user possesses the privileges *Select*, *Reload*, and *Super*.

```
LOAD INDEX INTO CACHE indexspec1, indexspec2 …
```

This command loads all specified indexes of MyISAM tables into the cache. This makes sense only in rare cases, such as carrying out repeatable benchmark tests.

indexspec: Specifies the MyISAM tables to be loaded. The following syntax is used:

tablename [[INDEX|KEY] (indexname1 …)] [IGNORE LEAVES]

Currently, the command loads all indexes of the table into the cache. Thus the specification of individual indexes will make sense only in the future. *IGNORE LEAVES* means that only a part of the index is to be loaded.

```
LOAD TABLE dbname.tablename FROM MASTER
```

This command copies a table in a replication system from master to slave, if the table does not yet exist there. The purpose of the command is actually to simplify debugging for MySQL developers. However, the command can possibly also be used for repairing a replication system after errors have been detected. The execution of the command requires that the replication user possess the privileges *Select*, *Reload*, and *Super*. *LOAD TABLE* works only for MyISAM tables.

```
LOCK TABLE table1 [ AS aliasname ] locktype, table2 [ AS alias2 ] locktype, …
```

LOCK TABLE prevents other MySQL users from executing write or read operations on the specified tables. If a table is already blocked by another user, then the command waits (unfortunately, without a timeout value, thus theoretically forever) until that block is released.

Table *LOCK*s ensure that during the execution of several commands no data are changed by other users. Typically, *LOCK*s are necessary when first a *SELECT* query is executed and then tables are changed with *UPDATE*, where the results of the previous query are used. (For a single *UPDATE* command, on the other hand, no *LOCK* is necessary. Individual *UPDATE* commands are always completely executed by the MySQL server without giving other users the opportunity to change data.)

LOCK TABLE should not be used on InnoDB tables, for which you can achieve much more efficient locking using transactions and the commands *SELECT … IN SHARE MODE* and *SELECT … FOR UPDATE*.

Note that *LOCK TABLE* commands end a running transaction as with *COMMIT*. For future versions of MySQL there are InnoDB-specific *LOCK* variants planned that will be able to be executed outside of transactions.

locktype: In MySQL 5.0 there are four *LOCK* types available:

READ: All MySQL users may read the table, but no one may change anything (including the user who executed the *LOCK* command). A *READ LOCK* is allocated only when the table is not blocked by other *WRITE LOCK*s. (Existing *READ LOCK*s, on the other hand, are no hindrance for new *READ LOCK*s. It is thus possible for several users to have simultaneous *READ LOCK*s on the same table.)

READ LOCAL: Like *READ*, except that *INSERT*s are allowed if they do not change existing data records.

WRITE: The current user may read and change the table. All other users are completely blocked. They may neither change data in the blocked table nor read it. A *WRITE LOCK* is allocated only if the table is not blocked by other *LOCK*s (*READ* or *WRITE*). Until the *WRITE LOCK* is lifted, other users can obtain neither a *READ LOCK* nor a *WRITE LOCK*.

LOW PRIORITY WRITE: Like *WRITE*, except that during the waiting time (that is, until all other *READ* and *WRITE LOCK*s have been ended) other users may obtain on demand a new *READ LOCK*. However, this means as well that the *LOCK* will be allocated only when there is no other user who wishes a *READ LOCK*.

In future versions of MySQL there will presumably be two additional *LOCK* variants especially for InnoDB tables. These two variants are not yet officially documented, and so it is possible that the following description is inaccurate.

IN SHARE MODE: This *LOCK* type will have the same effect as *SELECT * FROM table* and will protect the entire table from changes by other connections. (*INSERT*, *UPDATE*, and *DELETE* commands of other connections are thus blocked until the end of the lock.)

There are two advantages over the *SELECT* command: The InnoDB table driver can execute locking more efficiently, and the *LOCK* command is compatible with other database systems (Oracle, PostgreSQL, etc.).

IN EXCLUSIVE MODE: This *LOCK* type is even more restrictive and blocks *SELECT* commands of other connections if this option is executed with the option *LOCK IN SHARE MODE* or *LOCK FOR UPDATE*.

Table *LOCK*s can increase the speed with which several database commands can be executed one after the other (of course, at the cost that other users are blocked during this time).

MySQL manages table *LOCK*s by means of a thread, where each connection is associated with its own thread. Only one *LOCK* command is considered per thread. (But several tables may be included.) As soon as *UNLOCK TABLES* or *LOCK* is executed for any other table, then all previous locks become invalid.

For reasons of efficiency, it should definitely be attempted to keep *LOCK*s as brief as possible and to end them as quickly as possible by *UNLOCK*. *LOCK*s end automatically when the current process ends (that is, for example, when the connection between server and client is broken).

```
OPTIMIZE TABLE tablename
```

OPTIMIZE TABLE removes, since MySQL 4.1.3, unused storage space from MyISAM and InnoDB tables and ensures that associated data in a data record are stored together.

OPTIMIZE TABLE should be regularly executed for tables whose contents are continually being changed (many *UPDATE* and *DELETE* commands). This speeds up data access. With MyISAM tables the database file is also made smaller. The *table space* of InnoDB tables, on the other hand, cannot in principle be made smaller.

```
PROCEDURE procname
```

MySQL can be extended with external procedures. Their code must be formulated in the C++ programming language. To use such functions in *SELECT* commands, the key word *PROCEDURE* must be used.

As an example of such a procedure, the MySQL program code contains the function *ANALYSE*. This procedure can be used to analyze the contents of a table in the hope of determining a better table definition. The function is called thus:

```
SELECT * FROM tablename PROCEDURE ANALYSE()
```

As with the creation of user-defined functions (UDFs; see *CREATE FUNCTION*), the programming of procedures requires a great deal of MySQL background knowledge. Further information can be found in the MySQL documentation: `http://dev.mysql.com/doc/mysql/en/extending-mysql.html`.

An elegant and much simpler alternative to UDFs is stored procedures, which have been available since MySQL 5.0.

```
PURGE MASTER LOGS TO 'hostname-bin.n'
```

This command deletes all binary logging files that are older than the file specified. Execute this command only when you are sure that the logging files are no longer needed, that is, when all slave computers have synchronized their databases. This command can be executed only on the master computer of a replication system, and only if the *Super* privilege has been granted. See also *RESET MASTER*.

```
RENAME TABLE oldtablename TO newtablename
```

RENAME TABLE gives a new name to an existing table. It is also possible to rename several tables, e.g., *a TO b*, *c TO d*, etc.

There is no command for giving a new name to an entire database. If you are using MyISAM tables, then to do so, you can stop the MySQL server, rename the database directory and then restart the server. Note that you may have to change access privileges in the *mysql* database. With InnoDB tables, you must make a backup (`mysqldump`) and then import the tables into a new database.

```
REPAIR TABLE tablename1, tablename2, … [ TYPE = QUICK ]
```

REPAIR TABLE attempts to repair a defective table file. With the option *TYPE = QUICK* only the index is re-created.

REPAIR TABLE can be used only with MyISAM tables. Instead of this command, you may also use the external program `myisamchk -r tblfile`. (If *REPAIR TABLE* does not return OK as result, then you might try `myisamchk -o`. This program offers more repair possibilities than *REPAIR TABLE*.)

```
REPLACE [INTO]
```

REPLACE is a variant of *INSERT*. The only difference relates to new records whose key word is the same as that of an existing record. In this case, the existing record is deleted and the new one is stored in the table. Since the behavior with duplicates is so clearly defined, *REPLACE* does not have the *IGNORE* option possessed by the *INSERT* command.

```
RESET MASTER
```

This command deletes all binary logging files including the index file `hostname-bin.index`. With this command, replication can be restarted at a particular time. For this, *RESET SLAVE* must be executed on all slave systems. Before the command is executed it must be ensured that the databases on all slave systems are identical to those of the master system. This command assumes the *reload* privilege.

If you wish to delete only old (no longer needed) logging files, then use *PURGE MASTER LOGS*.

```
RESET QUERY CACHE
```

This command deletes all entries from the query cache. It assumes the *reload* privilege.

```
RESET SLAVE
```

This command reinitializes the slave system. The contents of `master.info` (and with it the current logging file and its position) are deleted. The command assumes the *reload* privilege.

This command makes sense only if after some problems the databases are to be set up on the slave based on previous snapshots so that the slave system then can synchronize itself by replication, or when *RESET MASTER* was executed on the master system (so that all logging files are deleted there). In this case, first *SLAVE STOP* and then *SLAVE START* should be executed on the slave system.

```
RESTORE TABLE tblname FROM '/backup/directory'
```

RESTORE TABLE copies the files of the specified table from a backup directory into the data directory of the current database. *RESTORE TABLE* is the inverse of *BACKUP TABLE*.

BACKUP and *RESTORE* do not work for InnoDB tables. Both commands are considered *deprecated* and should not be used if possible. Alternatives are external backup tools such as `mysqldump`, `mysqlhotcopy`, and the InnoDB backup utility.

```
REVOKE privileges ON objects FROM users
```

REVOKE is the inverse of *GRANT*. With this command you can remove individual privileges previously granted. The syntax for the parameters *privileges*, *objects*, and *users* can be read about under the *GRANT* command. The only difference relates to the *Grant* privilege: To revoke this privilege from a user, *REVOKE* can be used in the following form: *REVOKE GRANT OPTION ON … FROM …*.

Although *GRANT* inserts new users into the *mysql.user* table, *REVOKE* is incapable of deleting this user. You can remove all privileges from this user with *REVOKE*, but you cannot prevent this user from establishing a connection to MySQL. (If you wish to take that capability away as well, you must explicitly remove the entries from the *user* database with the *DELETE* command.)

Please note that in the MySQL access system you cannot forbid what is allowed at a higher level. If you allow *x* access to database *d*, then you cannot exclude table *d.t* with *REVOKE*. If you wish to allow *x* access to all tables of the database *d* with the exception of table *t*, then you must forbid access to the entire database and then allow access to individual tables of the database (with exception of *d*). *REVOKE* is not smart enough to carry out such operations on its own.

```
ROLLBACK
```

ROLLBACK undoes the most recent transaction. (Instead of *ROLLBACK*, you can confirm the pending changes with *COMMIT* and thereby finalize their execution.) *BEGIN/COMMIT/ROLLBACK* work only if you are working with transaction-capable tables. Further information and examples on transactions can be found in Chapter 10.

```
ROLLBACK TO SAVEPOINT name
```

ROLLBACK ends the current transaction. All SQL commands executed since the specified *SAVEPOINT* are canceled. Commands before the *SAVEPOINT* are accepted. The command has been available since version 4.0.14.

```
SAVEPOINT name
```

This command sets, since MySQL version 4.0.14, a mark within the running transaction. With *ROLLBACK TO SAVEPOINT* the transaction can be aborted beyond this point. *SAVEPOINT*s are valid only within a transaction. They are deleted at the end of the transaction.

```
SELECT [selectoptions] column1 [[AS] alias1], column2 [[AS] alias2] ...
  [ FROM tablelist ]
  [ WHERE condition ]
  [ GROUP BY groupfield  [ASC |DESC] ]
  [ HAVING condition ]
  [ ORDER BY ordercolumn1 [DESC], ordercolumn2 [DESC] ... ]
  [ LIMIT [offset,] rows ]
  [ PROCEDURE procname]
  [ LOCK IN SHARE MODE  |   FOR UPDATE ]
```

SELECT serves to formulate database queries. It returns the query result in tabular form. *SELECT* is usually implemented in the following form:

```
SELECT column1, column2, column3 FROM table ORDER BY column1
```

However, there are countless syntactic variants, thanks to which *SELECT* can be used also, for example, for processing simple expressions.

```
SELECT HOUR(NOW( ))
```

Note that the various parts of the *SELECT* command must be given in the order presented here.

selectoptions: The behavior of this command can be controlled by a number of options:

DISTINCT | *DISTINCTROW* specify how MySQL should behave when a query returns several identical records. *DISTINCT* and *DISTINCTROW* mean that identical result records should be displayed only once. *ALL* means that all records should be displayed (the default setting).

Since MySQL 4.1, the sort order can also be specified for *DISTINCT* (which is also the basis for determining equivalence of character strings): The syntax is *DISTINCT column COLLATE collname*.

HIGH_PRIORITY has the effect that a query with higher priority than change or insert commands will be executed. *HIGH_PRIORITY* should be used only for queries that need to be executed very quickly.

SQL_SMALL_RESULT | *SQL_BIG_RESULT* specify whether a large or small record list is expected as result, and they help MySQL in optimization. Both options are useful only with *GROUP BY* and *DISTINCT* queries.

SQL_BUFFER_RESULT has the effect that the result of a query is stored in a temporary table. This option should be used when the evaluation of the query is expected to range over a long period of time and locking problems are to be avoided during this period.

SQL_CACHE and *SQL_NO_CACHE* specify whether the results of the *SELECT* command should be stored in the query cache or whether such storage should be prevented. By default, *SQL_CACHE* usually holds, unless the query cache is executed in demand mode (*QUERY_CACHE_TYPE=2*).

SQL_CALC_FOUND_ROWS has the effect that MySQL determines the total number of found records even if you limit the result with *LIMIT*. The number can then be determined with a second query *SELECT FOUND_ROWS( )*.

STRAIGHT_JOIN has the effect that data collected from queries extending over more than one table should be joined in the order of the *FROM* expression. (Without *STRAIGHT_JOIN*, MySQL attempts to find the optimal order on its own. *STRAIGHT_JOIN* bypasses this optimization algorithm.)

column: Here column names are normally given. If the query encompasses several tables, then the format is *table.column*. If a query is to encompass all the columns of the tables specified by *FROM*, then you can save yourself some typing and simply specify *. (Note, however, that this is inefficient in execution if you do not need all the columns.)

However, instead of column names, you may also use general expressions or functions, e.g., for formatting a column (*DATE_FORMAT(...)*) or for calculating an expression (*COUNT(...)*).

With *AS*, a column can be given a new name. This is practical in using functions such as *HOUR(column) AS hr*. The use of *AS* is optional. That is, *HOUR(column) hr* is syntactically correct (but not as readable).

Such an alias name can then be used in most of the rest of the *SELECT* command (e.g., *ORDER BY hr*). The alias name cannot, however, be placed in a *WHERE* clause.

Since MySQL 4.1, the desired sort order can be specified with *COLLATE* (e.g., *column COLLATE collname AS alias*).

tablelist: In the simplest case, there is simply a list (separated by commas) of all tables that are to be considered in the query. If no relational conditions (further below with *WHERE*) are formulated, then MySQL returns a list of all possible combinations of data records of all affected tables.

There is the possibility of specifying here a condition for linking the tables, for example in the following forms:

table1 LEFT [OUTER] JOIN table2 ON table1.xyID = table2.xyID

table1 LEFT [OUTER] JOIN table2 USING (xyID)

table1 NATURAL [LEFT [OUTER]] JOIN

An extensive list of the many syntactic synonyms can be found in Chapter 9, where the topic of links among several tables is covered in great detail.

Within *tablelist* an alias can be given for every table name. The key word *AS* is optional:

table1 [AS] t1, table2 [AS] t2

condition: Here is where conditions that the query results must fulfill can be formulated. Conditions can contain comparisons (*column1>10* or *column1=column2*) or pattern expressions (*column LIKE '%xy'*), for example. Several conditions can be joined with *AND*, *OR*, and *NOT*.

MySQL allows selection conditions with *IN*:

WHERE id IN(1, 2, 3) is equivalent to *WHERE id=1 OR id=2 OR id=3*.

WHERE id NOT IN (1,2) is equivalent to *WHERE NOT (id=1 OR id=2)*.

Full-text search: Conditions can also be formulated with M*ATCH(col1, col2) AGAINST('word1 word2 word3')*. Thereby a full-text search is carried out in the columns *col1* and *col2* for the words *word1*, *word2*, and *word3*. (This assumes that a full-text index for the columns *col1* and *col2* has been created.)

AGAINST also supports Boolean search expressions, for example, in the form *AGAINST('+word1 +word2 -word3' IN BOOLEAN MODE*). Here the plus sign represents a logical AND operation, while the minus sign means that the specified word may not appear in the record. The full syntax for search expressions can be found in Chapter 10.

***WHERE* versus *HAVING*:** Conditions can be formulated with *WHERE* or *HAVING*. *WHERE* conditions are applied directly to the columns of the tables named in *FROM*.

HAVING conditions, on the other hand, are applied only after the *WHERE* conditions to the intermediate result of the query. The advantage of *HAVING* is that conditions can also be specified for function results (for example, *SUM(column1)* in a *GROUP BY* query). Alias names can be used in *HAVING* conditions (*AS xxx*), which is not possible in *WHERE*.

Conditions that can be equally well formulated with *WHERE* or *HAVING* should be expressed with *WHERE*, because in that case, better optimization is possible.

groupfield: With *GROUP BY*, you can specify a group column. If the query returns several records with the same values for the group column, then these records are collected into a single new record. Along with *GROUP BY*, in the *column* part of the query so-called aggregate functions are usually placed, with which calculations can be made over grouped fields (e.g., *COUNT*, *SUM*, *MIN*, *MAX*).

By default, grouped results are sorted as though *ORDER BY* had been specified for the columns. Optionally, the sort order can be determined with *ASC* or *DESC*.

Since MySQL 4.1, the desired sort order can be given with *COLLATE* (e.g., *GROUP BY column COLLATE collname*).

ordercolumn: With *ORDER BY* several columns or expressions can be specified according to which the query result should be sorted. Sorting normally proceeds in increasing order (A, B, C, … or 1, 2, 3, …). With the option *DESC* (for *descending*) you have decreasing order.

[offset,] row: With *LIMIT* the query results can be reduced to an arbitrary selection. This is to be recommended especially when the results are to be displayed pagewise or when the number of result records is to be limited. The position at which the results are to begin is given by *offset* (0 for the first data record), while *row* determines the maximum number of result records.

procname: This enables the call of a user-defined procedure (see the description of *PROCEDURE*).

MySQL does not support the formulation *SELECT … INTO table*, which is recognized in many other SQL dialects. In most cases you can get around this lack with *INSERT INTO … SELECT …* or *CREATE TABLE tablename … SELECT …*.

Locking (InnoDB Tables)

If you use transactions, then the addition of *LOCK IN SHARE MODE* has the effect that all records found by the *SELECT* command will be blocked by a *shared lock* until the end of the transaction. This has two consequences: First, your *SELECT* query will not be executed until there is no running transaction that could change the result. Second, the affected records cannot now be changed by other connections (though they can be read with *SELECT*) until your transaction has ended.

FOR UPDATE is even more restrictive, blocking the found records with an *exclusive lock*. In contrast to a *shared lock*, other connections that execute *SELECT … LOCK IN SHARE MODE* must now wait until the end of your transaction (of course, only if the same records are affected by *SELECT* queries).

SubSELECTs

Beginning with version 4.1, MySQL supports so-called *subSELECT*s. This means that the results of one query can flow into a condition of another. Examples of the use of *subSELECT*s can be found in Chapter 10. Here are the syntactic variants:

SELECT … WHERE col = [ANY|ALL|SOME] (SELECT …)

In this variant the second *SELECT* query must return a single value (one row and one column). This value is used for the comparison *col* = …. Other comparison operators are allowed as well, such as *col* > … and *col* <= … and *col* <> ….

SELECT … WHERE col = ANY|SOME (SELECT …)

Here the second *SELECT* query can return more than one value. *ANY* or the equivalent key word *SOME* results in all suitable values being considered. Then the entire query returns more than one result. *col = ANY* … is the same as col *IN* … (see below).

SELECT … WHERE col = ALL (SELECT …)

The expression *comparisonoperator ALL* has the value *TRUE* if the comparison is true for all the results of the second *SELECT* query or if the second *SELECT* query returns no results at all.

SELECT … WHERE col [NOT] IN (SELECT …)

With this variant, the second *SELECT* query can return a list of individual values. This list is then processed in the form *SELECT ... WHERE col IN (n1, n2, n3)*. In place of *IN* one may also use *NOT IN*.

SELECT ROW(value1, value2 …) = [ANY] (SELECT col1, col2 …)

This query tests whether there exists a record that satisfies certain criteria. The result can be either 1 (true) or *NULL* (false). As comparison criterion there is not a single value, but a group of values. If the result record agrees with the *ROW* record, the entire query returns 1, otherwise *NULL*.

If the optional key word *ANY* or its synonym *SOME* is used, the second *SELECT* query can return more than one result. If at least one of these agrees with the *ROW* record, then the entire query returns 1.

SELECT … WHERE [NOT] EXISTS (SELECT …)

With this variant, the second query is executed for each record found from the first *SELECT* query. Only if this returns a result (at least one record) does the record from the first *SELECT* query remain in the result list. *EXISTS* constructions are as a rule useful only if the records of the two *SELECT* commands are linked with a *WHERE* condition.

SELECT … FROM (SELECT …) AS name WHERE …

With this variant first the *SELECT* command in parentheses is executed. It returns a table that serves as the basis for the outer *SELECT* query. The outer *SELECT* command thus does not access a preexisting table, but instead the table that was the result of the previous *SELECT* command. Such tables are called *derived tables*. The SQL syntax prescribes that these tables must be named using *AS name*.

SELECT commands can be nested within one another. Such commands are difficult to read and understand, however. *SELECT* commands can also be placed in the *WHERE* condition of *UPDATE* or *DELETE* commands to decide which records should be altered or deleted.

```
SELECT [selectoptions] columnlist
 INTO @var1, @var2 …
 [ FROM … WHERE … GROUP BY … HAVING … ORDER BY  … LIMIT … ]
```

With this variant of the *SELECT* command, all columns of the result record are stored, since MySQL 5.0, in the variables *var1*, *var2*, etc. This works only if the query returns exactly one record. (If necessary use *LIMIT 1*.)

INTO @var1, @var2 … can also be placed at the end of the instruction. The following two examples are therefore equivalent:

```
SELECT title, subtitle INTO @mytitle, @mysub FROM titles WHERE titleID=1
SELECT title, subtitle FROM titles WHERE titleID=1 INTO @mytitle, @mysub
```

```
SELECT [selectoptions] columnlist
  INTO OUTFILE 'filename' exportopt
  [ FROM … WHERE … GROUP BY … HAVING … ORDER BY  … LIMIT … ]
```

With this variant of the *SELECT* command, the records are written into a text file. Here we describe only those options that are specific to this variant. All other points of syntax can be found under *SELECT*.

filename: The file is generated in the file system of the MySQL server. For security reasons, the file should not already exist. Moreover, you must have the *FILE* privilege to be able to execute this *SELECT* variant.

exportoptions: Here it is specified how the text file is formatted. The entire option block looks like this:

[FIELDS

[TERMINATED BY 'fieldtermstring']

[[OPTIONALLY] ENCLOSED BY 'enclosechar']

[ESCAPED BY 'escchar']]

[LINES TERMINATED BY 'linetermstring']

fieldtermstring specifies the character string that separates columns within a line (e.g., a tab character).

enclosechar specifies a character that is placed before and after every entry, e.g., '123' with *ENCLOSED BY ' \"*. With *OPTIONALLY*, the character is used only on *CHAR*, *VARCHAR*, *TEXT*, *BLOB*, *TIME*, *DATE*, *SET*, and *ENUM* columns (and not for every number format, such as *TIMESTAMP*).

escchar specifies the character to be used to mark special characters (usually the backslash). This is especially necessary when in character strings of a text file special characters appear that are also used for separating data elements.

If *escchar* is specified, then the escape character is always used for itself (\\) as well as for ASCII code 0 (\0). If *enclosechar* is empty, then the escape character is also used as identifier of the first character of *fieldtermstring* and *linetermstring* (e.g., *t* and *n*). On the other hand, if *enclosechar* is not empty, then *escchar* is used only for *enclosechar* (e.g., \"), and not for *fieldtermstring* and *linetermstring*. (This is no longer necessary, since the end of the character string is uniquely identifiable due to *enclosechar*.)

linetermstring specifies the character string with which lines are to be terminated. With DOS/Windows text files this must be the character string ' *r**n*'.

In the four character strings, special characters can be specified, for example, *b* for backspace. The list of permissible special characters can be found at the command *LOAD DATA*. Moreover, character strings can be given in hexadecimal notation (such as *0x22* instead of ' \").

As with *LOAD DATA* the following is the default setting:

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

LINES TERMINATED BY '\n'

If you wish to input files generated with *SELECT … INTO OUTFILE* again into a table, then use *LOAD DATA*. This command is the inverse of *SELECT … INTO OUTFILE*. Further information and concrete application examples for both commands can be found in Chapter 14.

```
SELECT [selectoptions] column
  INTO DUMPFILE 'filename'
  [ FROM … WHERE … GROUP BY … HAVING … ORDER BY  … LIMIT … ]
```

SELECT … INTO DUMPFILE has, in principle, the same function as *SELECT … INTO OUTFILE* (see above). The difference is that here data are stored without any characters to indicate column or row division.

SELECT … INTO DUMPFILE is designed for saving a single BLOB object into a file. The *SELECT* query should therefore return precisely one column and one row as result. Should that not be the case, that is, if the query returns more than one data element, then usually (and for some strange reason not always) one receives an error message: *ERROR 1172*: *Result consisted of more than one row.*

```
(SELECT selectoptions) UNION [ALL] (SELECT selectoptions) unionoptions
```

Since MySQL 4.0 you can use *UNION* to unite the results of two or more *SELECT* queries. You thereby obtain a result table in which the results of the individual queries are simply strung together. The individual queries can affect different tables, though you must ensure that the number of columns and their data types are the same.

The optional key word *ALL* has the effect that duplicates (that is, results that arise in more than one *SELECT* query) appear in the end result with their corresponding multiplicity. Without *ALL*, duplicates are eliminated (as with *DISTINCT* in *SELECT*).

With *SELECT* commands, all options described earlier for selecting columns, setting sort order, etc., are permitted. With *unionoptions* you can also specify how the final result is to be sorted (*ORDER BY*) and reduced (*LIMIT*).

```
SET @variable1 = expression1, @variable2 = expression2 …
```

MySQL permits the management of one's own user variables. These variables are indicated by the @ symbol before the name. These variables are managed separately for each client connection, so that no naming conflicts can arise among clients. The content of such variables is lost at the end of the connection.

Instead of *SET*, one may also use *SELECT* for the assignment of user variables. The syntax is *SELECT @variable:=expression* (note that := must be used instead of =).

```
SET [options] [@@]systemvariable = expression
```

If the variable name has either no @ prefixed or two of them (@@), then *SET* is setting system variables.

> *options*: MySQL distinguishes two levels of validity among system variables: *GLOBAL* (valid for the entire MySQL server) and *SESSION* (valid only for the current connection). The default setting is *SESSION*.
>
> Instead of the options *GLOBAL* and *SESSION* you can prefix *global.* or *session.* to the variable names. Thus *SET GLOBAL name* = … is equivalent to *SET global.name* = ….

Variables at the global level can be changed only by users with the *Super* privilege. Global changes are valid only for new connections, not those already in existence.

```
SET [OPTION] option=value
```

SET can also be used to modify certain MySQL options as well as the password. Although the syntax looks the same as that for variable assignment, most of the options described here cannot be evaluated with *SHOW VARIABLES*. However, *SELECT @@name* works in most cases.

For example, with

```
SET SQL_LOW_PRIORITY_UPDATES = 0 / 1
```

it is possible to determine the order in which MySQL executes queries and change commands. The default behavior (1) gives priority to change commands. (This has the effect that a lengthy *SELECT* command will not block change commands, which are usually executed quickly.) With the setting 0, on the other hand, changes are executed only when no *SELECT* command is waiting to be executed.

Important SET Options

Here are the most important *SET* syntax variants, presented in alphabetical order:

SET AUTOCOMMIT = 0 or *SET AUTOCOMMIT = 1* switches the autocommit mode for transactions off or on. Autocommit mode holds only for transaction-capable tables (see Chapter 10).

SET CHARACTER SET 'csname' assigns the character set *csname* to the two session variables *character_set_client* and *character_set_result* and to the session variable *character_set_connection* the value of *character_set_database*. *SET CHARACTER SET DEFAULT* resets the three variables to their default settings. A description of *character_set_xxx* variables appears in Chapter 10.

SET FOREIGN_KEY_CHECKS = 0 or *1* deactivates or activates the checking of foreign key constraints (see also Chapter 8).

If you use replication, you can temporarily interrupt binary logging on the master system with *SET SQL_LOG_BIN =0* in order to make manual changes that should not be replicated. *SET SQL_LOG_BIN=1* resumes logging.

SET NAMES 'csname' is a variant of *SET CHARACTER SET*. The difference is that the character set *csname* is assigned to all three session variables *character_set_client*, *character_set_result*, and *character_set_connection*.

SET PASSWORD offers a convenient way of changing one's own password, sparing the comparatively difficult manipulation of the access tables in the *mysql* database:

```
SET PASSWORD = PASSWORD('some password')
```

If you have sufficient privileges, you can also set another user's password with *SET*.

```
SET PASSWORD FOR username@hostname = PASSWORD('newPassword')
```

SET SQL_QUERY_CACHE = 0|1|2|ON|OFF|DEMAND sets the mode of the query cache (see Chapter 14).

SET TRANSACTION ISOLATION LEVEL sets the isolation level for transactions. The setting holds for transaction-capable tables. Here is the syntax:

```
SET [SESSION|GLOBAL] TRANSACTION ISOLATION LEVEL
    READ UNCOMMITTED | READ COMMITTED |
    REPEATABLE READ | SERIALIZABLE
```

SET SESSION changes the transaction degree for the current connection, and *SET GLOBAL* for all future connections (but not the current one). If neither *SESSION* nor *GLOBAL* is specified, then the setting is valid only for the coming transaction. (Note that *SESSION* and *GLOBAL* in *SET TRANSACTION* have a somewhat different effect from that of *SET [@@]systemvariable*.)

The four isolation degrees are described in Chapter 10. With InnoDB tables, the default is *REPEATABLE READ*. The isolation degree can also be read from the variable *@@[global.]tx_isolation*.

Additional SET Options

The following list presents all the options that can be changed with *SET*:

```
SET BIG_TABLES = 0 | 1
SET CHARACTER SET character_set_name | DEFAULT
SET IDENTITY = #
SET INSERT_ID = #
SET LAST_INSERT_ID = #
SET LOW_PRIORITY_UPDATES = 0 | 1
SET MAX_JOIN_SIZE = value | DEFAULT
SET QUERY_CACHE_TYPE = 0 | 1 | 2
SET QUERY_CACHE_TYPE = OFF | ON | DEMAND
SET SQL_AUTO_IS_NULL = 0 | 1
SET SQL_BIG_SELECTS = 0 | 1
SET SQL_BUFFER_RESULT = 0 | 1
SET SQL_LOG_OFF = 0 | 1
SET SQL_LOG_UPDATE = 0 | 1
SET SQL_QUOTE_SHOW_CREATE = 0 | 1
SET SQL_SAFE_UPDATES = 0 | 1
SET SQL_SELECT_LIMIT = value | DEFAULT
SET TIMESTAMP = timestamp_value | DEFAULT
```

An explanation of these (mostly seldom used) setting options can be found in the MySQL documentation in the description of the *SET* command: `http://dev.mysql.com/doc/mysql/en/set-option.html`.

```
SHOW BINLOG EVENTS [IN logname] [FROM pos] [LIMIT offset, rows]
```

If this command is executed without options, then it returns the complete contents of the currently active binary logging file. The options allow for the specification of other logging files or for limiting the output. Note that this command can also be used to read the logging file of an external MySQL server.

```
SHOW CHARACTER SET [ LIKE pattern ]
```

Since MySQL 4.1, *SHOW CHARACTER SET* returns a list of all available character sets and their default sort orders.

```
SHOW COLLATION[ LIKE pattern ]
```

Since MySQL 4.1, *SHOW COLLATION* returns a list of all available sort orders.

```
SHOW COLUMN TYPES
```

In the future, *SHOW COLUMN TYPES* will return a list of all data types available for column definition in a table. (In the tested version 5.0.2 the resulting list was incomplete.)

```
SHOW [FULL] COLUMNS FROM tablename
    [ FROM databasename ]   [ LIKE pattern ]
```

SHOW COLUMNS returns a table with information on all columns of a table (field name, field type, index, default value, etc.). With *LIKE* the list of columns can be filtered with a search pattern with the wild cards _ and %. The optional key word *FULL* has the effect that the access privileges of the current user are also displayed on the columns. The same information can be obtained with *SHOW FIELDS FROM tablename*, *EXPLAIN tablename*, or *DESCRIBE tablename*, as well as with the external program `mysqlshow`. More detailed information can be found in the virtual table *information_schema.columns* (since MySQL 5.0).

```
SHOW CREATE DATABASE tablename
```

Since MySQL 4.1, *SHOW CREATE DATABASE* displays the SQL command with which the specified database can be re-created.

```
SHOW CREATE FUNCTION/PROCEDURE name
```

SHOW CREATE FUNCTION/PROCEDURE displays since MySQL 5.0 the SQL command with which the specified stored procedure can be re-created.

```
SHOW CREATE TABLE name
```

SHOW CREATE TABLE displays the SQL command with which the specified table or view can be re-created.

In the result of the command, all object names are set between backward single quotes, for example *`tablename`* or *`columnname`*. If you don't want this, then execute *SET SQL_QUOTE_SHOW_CREATE=0* beforehand.

```
SHOW CREATE VIEW name
```

SHOW CREATE VIEW displays the SQL command with which the specified view can be re-created. The command assumes the *Create View* privilege.

```
SHOW DATABASES [ LIKE pattern ]
```

SHOW DATABASES returns a list of all databases that the user can access. The list can be filtered with a search pattern with the wild cards _ and %. The same information can also be obtained with the external program `mysqlshow`.

For users possessing the *Show Databases* privilege, *SHOW DATABASES* returns a list of all databases, including those to which the user does not have access.

```
SHOW [STORAGE] ENGINES
```

Since MySQL 4.1, *SHOW ENGINES* displays a list of all table drivers (MyISAM, InnoDB, etc.) including information as to whether the driver is supported by the current MySQL version.

```
SHOW [COUNT(*)] ERRORS [LIMIT [offset, ] count]
```

Since MySQL 4.1, *SHOW ERRORS* returns a list of errors that were triggered by the execution of the most recent command. With *LIMIT* the result can be limited as with a *SELECT* command.

```
SHOW FIELDS
```

See *SHOW COLUMNS*.

```
SHOW FUNCTION STATUS [LIKE 'pattern']
```

This command returns since MySQL 5.0 a list of all functions (SPs). The list covers all databases. Optionally, the list can be reduced to all functions whose names satisfy a search pattern (where the SQL wild cards % and _ are allowed). A list of all procedures can be determined with *SHOW PROCEDURE STATUS*.

```
SHOW GRANTS FOR user@host
```

SHOW GRANTS displays a list of all access privileges for a particular user. It is necessary that *user* and *host* be specified exactly as these character strings are stored in the various *mysql* access tables. Wild cards are not permitted.

```
SHOW INDEX FROM table
```

SHOW INDEX returns a table with information about all indexes of the given table.

```
SHOW INNODB STATUS
```

SHOW INNODB STATUS returns information about various internal workings of the InnoDB table driver. The data can be used for speed optimization. More information can be found at `http://dev.mysql.com/doc/mysql/en/innodb-tuning.html`.

```
SHOW KEYS
```

See *SHOW INDEX*.

```
SHOW [BDB] LOGS
```

This command shows which BDB logging files are currently being used. (If you are not using BDB tables, the command returns no result.)

```
SHOW MASTER LOGS
```

This command returns a list of all binary logging files. It can be executed only on the master computer of a replication system.

```
SHOW MASTER STATUS
```

This command shows which logging file is the current one, as well as the current position in this file and which databases are excepted from logging (configuration settings `binlog-do-db` and `binlog-ignore-db`). This command can be used only on the master computer of a replication system.

```
SHOW PRIVILEGES
```

Since MySQL 4.1, this command returns a list of all available privileges with a brief description.

```
SHOW PROCEDURE STATUS [LIKE 'pattern']
```

This command returns, since MySQL 5.0, a list of all procedures (SPs). The list comprises SPs from all databases. Optionally, the list can be reduced to all procedures whose names match the pattern *pattern* (where the SQL wild cards % and _ are allowed). A list of all functions can be determined with *SHOW FUNCTION STATUS*.

```
SHOW  [ FULL ]  PROCESSLIST
```

This command returns a list of all running threads (subprocesses) of the MySQL server. If the *PROCESS* privilege has been granted, then all threads are shown. Otherwise, only the user's threads are displayed.

The option *FULL* has the effect that for each thread, the complete text of the most recently executed command is displayed. Without this option, only the first 100 characters are shown.

The process list can also be determined with the external command `mysqladmin`.

```
SHOW SLAVE HOSTS
```

This command returns a list of all slaves that replicate the master's databases. The command can be used only by the master computer of a replication system. It functions only for slaves for which the host name is specified in the configuration file explicitly in the form `report-host = hostname`.

```
SHOW SLAVE STATUS
```

This command provides information on the state of replication, including the display of all information about the file `master.info`. This command can be executed only on a slave computer in a replication system.

```
SHOW STATUS
```

This command returns a list of various MySQL variables that provide information on the current state of MySQL (for example, *Connections*, *Open_files*, *Uptime*). This same information can also be determined with the external program `mysqladmin`. A description of all variables can be found in the MySQL documentation under the command *SHOW STATUS*: `http://dev.mysql.com/doc/mysql/en/show-status.html`.

```
SHOW TABLE STATUS  [FROM database] [LIKE pattern]
```

SHOW TABLE STATUS returns information about all tables of the currently active or specified database: table type, number of records, average record length, *Create_time*, *Update_time*, etc. The same information can also be determined with the external program `mysqlshow`. With *pattern* the list of tables can be limited; the SQL wild cards % and _ are permitted in *pattern*.

```
SHOW TABLE TYPES see also SHOW ENGINES
```

Since MySQL 4.1, *SHOW TABLE TYPES* returns a list of all available table types (MyISAM, HEAP, InnoDB, etc.).

```
SHOW TABLES [FROM database] [LIKE pattern]
```

SHOW TABLES returns a list of all tables and *Views* of the current (or specified) database. Optionally, the list of all tables can be reduced to those matching the search pattern *pattern* (where the SQL wild cards % and _ are allowed). More information on the construction of individual tables can be obtained with *DESCRIBE TABLE* and *SHOW COLUMNS*. The list of tables can also be retrieved with the external program `mysqlshow`.

```
SHOW [options] VARIABLES [LIKE pattern]
```

This command returns a seemingly endless list of all system variables defined by MySQL together with their values (e.g., *ansi_mode*, *sort_buffer*, *tmpdir*, *wait_timeout*, to name but a very few). To limit the list, a pattern can be given (e.g., *LIKE 'char%'*).

Many of these variables can be set at launch of MySQL or afterwards with *SET*. The list of variables can also be recovered with the external command `mysqladmin`.

options: Here you can specify *GLOBAL* or *SESSION*. *GLOBAL* has the effect that the default values valid at the global level are displayed. *SESSION*, on the other hand, results in the values being displayed that are valid for the current connection. The default is *SESSION*.

An extensive description of the variables can be found in the MySQL documentation: `http://dev.mysql.com/doc/mysql/en/server-system-variables.html`.

```
SHOW [COUNT(*)] WARNINGS [LIMIT [offset, ] count]
```

Since MySQL 4.1, this command returns a list of all warnings that arose from the execution of the most recent command.

```
SLAVE START/STOP [IO_THREAD | SQL_THREAD]
```

These commands start and stop replication (we leave it to the reader to determine which is which). They can be executed only on the slave computer of a replication system.

By default, two threads are started for replication: the IO thread (copies the binary logging data from the master to the slave) and the SQL thread (executes the logging file's SQL command). With the optional specification of *IO_THREAD* or *SQL_THREAD*, these two threads can be started or stopped independently (which makes sense only for debugging).

```
START TRANSACTION
```

If you are working with transaction-capable tables (InnoDB), *START TRANSACTION* initiates a new transaction. The command is ANSI-99 conforming, but it has been available only since MySQL 4.0.11. In older versions, you must use the equivalent command *BEGIN*.

```
TRUNCATE TABLE tablename
```

TRUNCATE has the same functionality as *DELETE* without a *WHERE* condition; that is, the effect is that all records in the table are deleted. This is accomplished by deleting the entire table and then re-creating it. (This is considerably faster than deleting each record individually.)

TRUNCATE cannot be part of a transaction. *TRUNCATE* functions like *COMMIT*; that is, all pending changes are first executed. *TRUNCATE* can also be undone with *ROLLBACK*.

```
UNION see also SELECT UNION
```

With *UNION*, you can assemble the results of several *SELECT* queries.

```
UNLOCK TABLES
```

UNLOCK TABLES removes all of the user's *LOCK*s. This command holds for all databases (that is, it doesn't matter which database is the current one).

```
UPDATE [updateoptions] tablename SET col1=value1, col2=value2 ..
  [ WHERE condition ]
  [ ORDER BY columns ]
  [ LIMIT maxrecords ]
```

UPDATE changes individual fields of the table records specified by *WHERE*. Those fields not specified by *SET* remain unchanged. In *value* one can refer to existing fields. For example, an *UPDATE* command may be of the following form:

```
UPDATE products SET price = price + 5 WHERE productID=3
```

Warning: Without a *WHERE* condition, all data records in the table will be changed. (In the above example, the prices of all products would be increased by 5.)

updateoptions: Here the options *LOW PRIORITY* and *IGNORE* may be given. The effect is the same as with *INSERT*.

condition: This condition specifies which records are affected by the change. For the syntax of *condition* see *SELECT*.

columns: With *ORDER BY*, you can sort the record list before making changes. This makes sense only in combination with *LIMIT*, for example, to change the first or last ten records (ordered according to some criterion). This possibility has existed since MySQL 4.0.

maxrecords: With *LIMIT*, the maximum number of records that may be changed is specified.

```
UPDATE [updateoptions] table1, table2, table3
  SET table1.col1=table2.col2 ...
  [ WHERE condition ] [ ORDER BY columns ] [ LIMIT maxrecords ]
```

Since MySQL 4.0, *UPDATE* commands can include more than one table. All tables included in the query must be specified after *UPDATE*. The only tables that are changed are those whose fields were specified by *SET*. The link between the tables must be set with *WHERE* conditions.

```
USE databasename
```

USE turns the specified database into the default database for the current connection to MySQL. Until the end of the connection (or until the next *USE* command), all table names are automatically assigned to the database *databasename*.

Function Reference

The functions described here can be used in *SELECT* queries as well as in other SQL commands. We begin with a few examples. In our first example, we shall join two table columns with *CONCAT* to create a new character string. In the second example, the function *PASSWORD* will be used to store an encrypted password in a column. In the third example, the function *DATE_FORMAT* will be summoned to help us format a date:

```
SELECT CONCAT(firstname, ' ', lastname) FROM users
   Peter Smith
   ...
INSERT INTO logins (username, userpassword)
VALUES ('smith', PASSWORD('xxx'))
SELECT DATE_FORMAT(a_date, '%Y %M %e')
FROM exceptions.test_date
    2005 December 7
```

Tip This section aims to provide only a compact overview of the functions available. Extensive information on these functions can be found in the MySQL documentation. Some of these functions have been introduced at various places in this book by way of example. See the Index for page numbers.

Arithmetic Functions

Arithmetic Functions	
ABS(x)	Calculates the absolute value (nonnegative number).
ACOS(x), *ASIN(x)*	Calculates the arcsin and arccos.
ATAN(x), *ATAN2(x, y)*	Calculates the arctangent.
CEILING(x)	Rounds up to the least integer greater than or equal to *x*.
COS(x)	Calculates the cosine; *x* is given in radians.
COT(x)	Calculates the cotangent.
DEGREES(x)	Converts radians to degrees (multiplication by 180/pi).
EXP(x)	Returns e^x.
FLOOR(x)	Rounds down to the greatest integer less than or equal to *x*.
LOG(x)	Returns the natural logarithm (i.e., to base *e*).
LOG10(x)&	Returns the logarithm to base 10.
MOD(x, y)	Returns the mod function, equivalent to *x % y*.
PI( )	Returns 3.1415927.
POW(x, y)	Returns x^y.
POWER(x, y)	Equivalent to *POW(x, y)*.
RADIANS(x)	Converts degrees into radians (multiplication by Pi/180).
RAND( )	Returns a random number between 0.0 and 1.0.
RAND(n)	Returns a reproducible (thus not quite random) number.
ROUND(x)	Rounds to the nearest integer.
ROUND(x, y)	Rounds to *y* decimal places.
SIGN(x)	Returns -1, 0, or 1 depending on the sign of *x*.

Arithmetic Functions	
SIN(x)	Calculates the sine.
SQRT(x)	Calculates the square root.
TAN(x)	Calculates the tangent.
TRUNCATE(x)	Removes digits after the decimal point.
TRUNCATE(x, y)	Retains *y* digits after the decimal point (thus T*RUNCATE(1.236439, 2)* returns 1.23).

In general, all functions return *NULL* if provided with invalid parameters (e.g., *SQRT(-1)*).

Comparison Functions, Tests, Branching

Comparison Functions	
COALESCE(x, y, z, ...)	Returns the first parameter that is not *NULL*.
GREATEST(x, y, z, ...)	Returns the greatest value or greatest character string.
IF(expr, val1, val2)	Returns *val1* if *expr* is true; otherwise, *val2*.
IFNULL(expr1, expr2)	Returns *expr2* if *expr1* is *NULL*; otherwise, *expr1*.
INTERVAL(x, n1, n2, ...)	Returns 0 if *x<n1*; 1 if *x< n2*, etc.; all parameters must be integers, and *n1 < n2 <* ... must hold.
ISNULL(x)	Returns 1 or 0, according to whether *x IS NULL* holds.
LEAST(x, y, z, ...)	Returns the smallest value or smallest character string.
STRCMP(s1, s2)	Returns 0 if *s1=s2* in sort order, -1 if *s1<s2*, 1 if *s1>s2*. Since MySQL 4.0 the function takes into account the valid character set. By default the function no longer distinguished uppercase and lowercase (which is different from MySQL 3.32).

Tests, Branching	
IF(expr, result1, result2)	Returns *result1* if *expr* is true; otherwise, *result2*.
CASE expr	Returns *result1* if *expr=val1*, returns *result2* if *expr=val2*.
WHEN val1 THEN result1 *WHEN val2 THEN result2* *ELSE resultn.*	If no condition is satisfied, then the result is *resultn*.
CASE *WHEN cond1 THEN result1* *WHEN cond2 THEN result2* *ELSE resultn* *END*	Returns *result1* if condition *cond1* is true, etc.

Type Conversion (Cast)

Type Conversion	
CAST(x AS type)	Changes *x* into the specified type. *CAST* works with the following types: *BINARY, CHAR, DATE, DATETIME, SIGNED [INTEGER], TIME,* and *UNSIGNED [INTEGER].*
CONVERT(x, type)	Equivalent to *CAST(x AS type).*
CONVERT(s USING cs)	Represents the string *s* in the character set *cs* (since MySQL 4.1).

String Processing

Most character string functions can also be used for processing binary data. Since MySQL 4.1 (with Unicode support), the position and length specification functions such as *LEFT* and *MID* apply to characters, not bytes. MID*(column, 3, 1)* thus returns the third character, regardless of the character set that is defined for *column*.

Processing Character Strings	
CHAR_LENGTH(s)	Returns the number of characters in *s*; *CHAR_LENGTH* works also for multibyte character sets (e.g., Unicode).
CONCAT(s1, s2, s3, …)	Concatenates the strings.
CONCAT_WS(x, s1, s2,…)	Functions like *CONCAT*, except that *x* is inserted between each string; *CONCAT_WS(', ', 'a', 'b', 'c')* returns *'a, b, c'*.
ELT(n, s1, s2, …)	Returns the *n*th string; *ELT(2, 'a', 'b', 'c')* returns *'b'*.
EXPORT_SET(x, s1, s2)	Creates a string from *s1* and *s2* based on the bit coding of *x*; *x* is interpreted as a 64-bit integer.
FIELD(s, s1, s2, …)	Compares *s* with strings *s1, s2* and returns the index of the first matching string; *FIELD('b', 'a', 'b', 'c')* returns 2.
FIND_IN_SET(s1, s2)	Searches for *s1* in *s2*; *s2* contains a comma-separated list of strings; *FIND_IN_SET('b', 'a,b,c')* returns 2.
INSERT(s1, pos, 0, s2)	Inserts *s2* into position *pos* in *s1*; *INSERT('ABCDEF', 3, 0, 'abc')* returns *'ABabcDEF'*.
INSERT(s1, pos, len, s2)	Inserts *s2* at position *pos* in *s1* and replaces *len* characters of *s2* with the new characters; *INSERT('ABCDEF', 3, 2, 'abc')* returns *'ABabcEF'*.
INSTR(s, sub)	Returns the position of *sub* in *s*; *INSTR('abcde', 'bc')* returns 2.
LCASE(s)	Changes uppercase characters to lowercase.
LEFT(s, n)	Returns the first *n* characters of *s*.
LENGTH(s)	Returns the number of bytes necessary to store the string *s*; if multibyte character sets are used (e.g., Unicode), then *CHAR_LENGTH* must be used to determine the number of characters.
LOCATE(sub, s)	Returns the position of *sub* in *s*; *LOCATE('bc', 'abcde')* returns 2.
LOCATE(sub, s, n)	As above, but the search for *sub* begins only at the *n*th character of *s*.
LOWER(s)	Transforms uppercase characters to lowercase.
LPAD(s, len, fill)	Inserts the fill character *fill* into *s*, so that *s* ends up with length *len*; *LPAD('ab', 5, '*')* returns *'***ab'*.
LTRIM(s)	Removes spaces at the beginning of *s*.
MAKE_SET(x, s1, s2 …)	Forms a new string in which all strings *sn* appear for which in *x* the bit *n* is set; *MAKE_SET(1+2+8, 'a', b', 'c', 'd')* returns *'a,b,d'*.
MID(s, pos, len)	Reads *len* characters from position *pos* from the string *s*; *MID('abcde', 3, 2)* returns *'cd'*.
POSITION(sub IN s)	Equivalent to *LOCATE(sub, s)*.
QUOTE(s)	Since MySQL 4.0, returns a string suitable for SQL commands; special characters such as ', ", \ are prefixed with a backspace.
REPEAT(s, n)	Joins *s* to itself *n* times; *REPEAT('ab', 3)* returns *'ababab'*.
REPLACE(s, fnd, rpl)	Replaces in *s* all *fnd* strings by *rpl*; *REPLACE('abcde', 'b', 'xy')* returns *'axycde'*.
REVERSE(s)	Reverses the string.
RIGHT(s, n)	Returns the last *n* characters of *s*.

Processing Character Strings	
RPAD(s, len, fill)	Inserts the fill character *fill* at the end of *s*, so that *s* has length *len*; *RPAD('ab', 5, '*')* returns *'ab***'*.
RTRIM(s)	Removes spaces from the end of *s*.
SPACE(n)	Returns *n* space characters.
SUBSTRING(s, pos)	Returns the right part of *s* from position *pos*.
SUBSTRING(s, pos, len)	As above, but only *len* characters (equivalent to *MID(s, pos, len)*).
SUBSTRING_INDEX(s, f, n)	Searches for the *n*th appearance of *f* in *s* and returns the left part of the string up to this position (exclusive); for negative *n*, the search begins at the end of the string, and the right part of the string is returned; *SUBSTRING_INDEX('abcabc', 'b', 2)* returns *'abca'* *SUBSTRING_INDEX('abcabc', 'b', -2)* returns *'cabc'*
TRIM(s)	Removes spaces from the beginning and end of *s*.
TRIM(f FROM s)	Removes the character *f* from the beginning and end of *s*.
UCASE(s) / UPPER(s)	Transforms lowercase characters to uppercase.

Converting Numbers and Character Strings	
ASCII(s)	Returns the byte code of the first character of *s*: thus *ASCII('A')* returns 65; see also *ORD*.
BIN(x)	Returns the binary code of *x*; *BIN(12)* returns *'1010'*.
CHAR(x, y, z, …)	Returns the string formed from the code *x, y, …*; *CHAR(65, 66)* returns *'AB'*.
CHARSET(s)	Since MySQL 4.1, returns the name of the character set in which *s* is represented.
CONV(x, from, to)	Transforms *x* from number base *from* to base *to*; *CONV(25, 10, 16)* returns the hexadecimal *'19'*.
CONVERT(s USING cs)	Since MySQL 4.1, represents the string *s* in the character set *cs*.
FORMAT(x, n)	Formats *x* with commas for thousands separation and *n* decimal places; *FORMAT(12345.678, 2)* returns *'12,345.68'*.
HEX(x)	Returns the hexadecimal code for *x*; *x* can be a 64-bit integer or (since MySQL 4.0) a character string; in the second case, each character is transformed into an 8-bit hex code; *HEX('abc')* returns *'414243'*.
INET_NTOA(n)	Transforms *n* into an IP address with at least four groups; *INET_NTOA(1852797041)* returns *'110.111.112.113' INET_ATON(ipadr)* transforms an IP address into the corresponding 32- or 64-bit integer; *INET_ATON('110.111.112.113')* returns 1852797041.
OCT(x)	Returns the octal code of *x*.
ORD(s)	Like *ASCII(s)*, returns the code of the first character, but functions also for multibyte character sets.
SOUNDEX(s)	Returns a string that should match similar-sounding English words; *SOUNDEX('hat')* and *SOUNDEX('head')* both return *'H300'*; extensive information on the SOUNDEX algorithm can be found in the book *SQL for Smarties* by Joe Celko.

Encryption of Character Strings and Password Management	
AES_DECRYPT(crypt, key)	Decrypts *crypt* with the AES algorithm (Rijndael) and uses *key* for decryption.
AES_ENCRYPT(str, key)	Encrypts *str* using *key* for encryption.
DES_ENCRYPT(str [, keyno | keystr])	Encrypts *str* using the DES algorithm; available since MySQL 4.0 and only when MySQL is compiled with SSL functions; without the optional second parameter, the first key from the DES key file is used for encryption; optionally, the number or name of the key can be specified.
DES_DECRYPT(crypt [, keyno | keystr])	Decrypts *crypt* using the DES algorithm.
DECODE(crypt, pw)	Decrypts *crypt* using the password *pw*.
ENCODE(str, pw)	Encrypts *str* using *pw* as password; the result is a binary object that can be decrypted with *DECODE*.
ENCRYPT(pw)	Encrypts the password with the UNIX *crypt* function; if this function is unavailable, returns *ENCRYPT NULL*.
PASSWORD(pw)	Encrypts the password with the algorithm that was used for storing passwords in the *USER* table; the result is a 16-character string; note that since MySQL 4.1, *PASSWORD* uses stronger encryption and returns a string of 45 characters.
OLD_PASSWORD(pw)	Encrypts the password as for *PASSWORD* under MySQL 3.23.*n* and 4.0.*n*; available since MySQL 4.1.

Calculation of Check Sums	
CRC32(s)	Since MySQL 4.1, computes a check (cyclic redundancy check value) for the string *s*.
MD5(str)	Computes the MD5 check sum for the string *str*.
SHA(str), SHA1(str)	Since MySQL 4.0, computes a 160-bit check sum using the SHA1 algorithm (defined in RFC 3174). SHA is considered more secure than MD5. The result is returned as a string containing a 40-digit hexadecimal code; SHA and SHA1 are synonyms.

Date and Time

Some of the following functions have been available only since MySQL 4.0 or 4.1.

Determining the Current Time	
CURDATE()	Returns the current date, e.g., *'2005-12-31'*. Synonym: *CURRENT_DATE()*
CURTIME()	Returns the current time as a string or an integer, depending on the context, e.g., *'23:59:59'* or *235959* (integer). Synonym: *CURRENT_TIME()*
NOW()	Returns the current time in the form *'2005-12-31 23:59:59'*. Synonyms: *CURRENT_TIMESTAMP()*, *LOCALTIME()*, *LOCALTIMESTAMP()*, *SYSDATE()*
UNIX_TIMESTAMP()	Returns the current system time as a Unix timestamp (32-bit integer).
UTC_DATE()	Returns the date in Coordinated Universal Time.
UTC_TIME()	Returns the time in Coordinated Universal Time.
UTC_DATETIME()	Returns the date and time in Coordinated Universal Time.

Calculating, Formatting, and Transformation Functions	
ADDDATE(d, n)	Adds *n* days to the starting time *d*.
ADDDATE(...)	Adds a time interval to the starting time *d* (see below). Synonym: *DATE_ADD()*
ADDTIME(d, t)	Adds the time *t* (*TIME*) to the starting time *d* (*DATETIME*).
CONVERT_TZ(d, tz1, tz2)	Converts the time *d* from the time zone *tz1* into the time zone *tz2*. The syntax for the given time zone depends on the operating system. Under Windows the time difference from UTC must be given (e.g., *'+2:00'*), while under Linux, after a configuration, you may use the name of the time zone (e.g., *'America/New_York'*; see Chapter 10).
DATE(d)	Returns only the date portion of a *DATETIME* expression. (That is, the function removes the time portion.)
DATEDIFF(d1, d2)	Returns the number of days between *d1* and *d2*. The time portion is ignored in the calculation.
DATE_FORMAT(d, form)	Formats *d* according to formatting string *f*; see below.
DAYNAME(date)	Returns *'Monday'*, *'Tuesday'*, etc.
DAYOFMONTH(date)	Returns the day of the month (1 to 31).
DAYOFWEEK(date)	Returns the day of the week (1 = Sunday through 7 = Saturday).
DAYOFYEAR(date)	Returns the day in the year (1 to 366).
EXTRACT(i FROM date)	Returns a number for the desired interval.
EXTRACT(YEAR FROM '2003-12-31')	Returns 2003.
FROM_DAYS(n)	Returns the date *n* days after the year 0.
FROM_DAYS(3660)	Returns *'0010-01-08'*.
FROM_UNIXTIME(t)	Transforms the Unix timestamp number *t* into a date.
FROM_UNIXTIME(0)	Returns *'1970-01-01 01:00:00'*.
FROM_UNIXTIME(t, f)	As above, but with formatting as in *DATE_FORMAT*.
GET_FORMAT(…)	Returns predefined formatting code for *DATE_FORMAT* (see the table further below).
HOUR(time)	Returns the hour (0 to 23).
LAST_DAY(d)	Returns the last day of the month specified by the date *d*. *LAST_DAY('2005-02-01')* returns *'2005-02-28'*.
MAKEDATE(y, dayofyear)	Creates a *DATE* expression from the input of year and day.
MAKETIME(h, m, s)	Creates a *TIME* expression from the input for hours, minutes, and seconds.
MICROSECOND(d)	Returns the number of microseconds (0 to 999999).
MINUTE(time)	Returns the minute (0 to 59).
MONTH(date)	Returns the month (1 to 12).
MONTHNAME(date)	Returns the name of the month (*'January'*, etc.).
PERIOD_ADD(s, n)	Adds *n* months to the start date, which must be specified in the form *'YYYYMM'* (e.g., *'200512'* for December 2005).
PERIOD_DIFF(s, e)	Returns the number of months between the start and end dates. Both times must be given in the form *'YYYYMM'*.
QUARTER(date)	Returns the quarter (1 to 4).

Continued

Calculating, Formatting, and Transformation Functions *(Continued)*	
SECOND(time)	Returns the second (0 to 59).
SEC_TO_TIME(n)	Returns the time *n* seconds after midnight.
SEC_TO_TIME(3603)	Returns *'01:00:03'*.
STR_TO_DATE(s, form)	Interprets the string *s* according to the formatting code in *form*. *STR_TO_DATE* is the inverse function of *DATE_FORMAT*.
SUBDATE(d, n)	Subtracts *n* days from the starting time *d*.
SUBDATE(d,)	Subtracts a time interval from the starting time *d* (see below). Synonym: *DATE_ADD()*
SUBTIME(d, t)	Subtracts the time *t* (*TIME*) from the starting time *d* (*DATETIME*).
TIMESTAMP(s)	Returns the starting time given in the string as a *TIMESTAMP* value. TIMESTAMP('2005-12-31') returns '2005-12-31 00:00:00'.
TIMESTAMP(s, time)	Returns *s* + *time* as a *TIMESTAMP* value.
TIMESTAMPADD(i, n, s)	Adds *n* times the interval *i* (e.g., *MONTH*) to the starting time *s*.
TIMESTAMPDIFF(i, s, e)	Returns the number of intervals *i* between the start time *s* and the end time *e*. *TIMESTAMPDIFF(HOUR, '2005-12-31', '2006-01-01')* returns 24.
TIME_FORMAT(time, f)	Like *DATE_FORMAT*, but for times only. *TIME_TO_SEC(time)* returns the number of seconds since midnight.
TO_DAYS(date)	Returns the number of days since the year 0.
UNIX_TIMESTAMP(d)	Returns the timestamp number for the given date.
WEEK(date)	Returns the week number (1 for the week beginning with the first Sunday in the year).
WEEK(date, mode)	Returns the week number (0 to 53 or 1 to 53). The parameter *mode* determines the first day of the week and how a week is defined. For example, *mode=0* means that weeks begin on Sunday and that 1 is the first week in the new year. If *mode* is not specified, the server default setting holds (*default_week_format* in `my.cnf/my.ini`). A description of modes can be found at `http://dev.mysql.com/doc/mysql/en/date-and-time-functions.html`.
WEEKDAY(date)	Returns the day of the week (0 = Monday, 1 = Tuesday, etc.).
WEEKOFYEAR(date)	Returns the calendar week (1 to 53).
YEAR(date)	Returns the year.
YEARWEEK(date, mode)	Returns an integer or string, depending on context, that consists of the year number and week number. *mode* is set as in *WEEK*. *YEARWEEK('2005-12-31')* returns *200552*.

The functions for dates and times generally assume that the initial data are valid. Do not expect a sensible return value for input such as *'2005-02-31'*.

With all functions that return a time or a date (or both), the format of the result depends on the context. Normally, the result is a character string (e.g., *'2005-12-31 23:59:59'*). However, if the function is used within a numeric calculation, for example *NOW()+0*, then the result is an integer of the form *20051231235959*.

Intervals with *ADDDATE*, *SUBDATE*, *EXTRACT*, *TIMESTAMPADD*, etc.

ADDDATE (date, INTERVAL n i) adds *n* times the interval *i* to the starting date *date*. The permitted interval names are collected in the table following some examples. The third example shows how intelligently the function deals with ends of months (31.12 or 28.2):

ADDDATE('2005-12-31', INTERVAL 3 DAY) returns *'2006-01-03'*.

ADDDATE('2005-12-31', INTERVAL '3:30' HOUR_MINUTE) returns *'2005-12-31 03:30:00'*.

ADDDATE ('2005-12-31', INTERVAL 2 month) returns '2006-02-28'.

SUBDATE is like *ADDDATE*, but it subtracts the given interval *n* times.

EXTRACT extracts the given interval from the time: *EXTRACT(MONTH FROM '2005-12-31')* returns 12.

TIMESTAMPADD is like *ADDDATE*, but it uses a different syntax for specifying the interval: *TIMESTAMPADD(DAY, 5, '2005-12-31')* returns *'2006-01-05'*.

TIMESTAMPDIFF returns the difference between two times as a multiple of the given interval: *TIMESTAMPDIFF(DAY, '2005-12-31', '2006-02-15')* returns 46.

Intervals for ADDDATE, SUBDATE, EXTRACT, TIMESTAMPADD	
MICROSECOND	*n*
SECOND	*n*
MINUTE	*n*
HOUR	*n*
DAY	*n*
MONTH	*n*
YEAR	*n*
SECOND_MICROSECOND	*'ss.mmmmmm'*
MINUTE_SECOND	*'mm:ss'*
HOUR_MINUTE	*'hh:mm'*
HOUR_SECOND	*'hh:mm:ss'*
DAY_HOUR	*'dd hh'*
DAY_MINUTE	*'dd hh:mm'*
DAY_SECOND	*'dd hh:mm:ss'*
YEAR_MONTH	*'yy-mm'*

Formatting Dates and Times

DATE_FORMAT(date, format) helps in representing dates and times in other formats than the usual MySQL format. Two examples illustrate the syntax:

DATE_FORMAT('2003-12-31', '%M %d %Y') returns *'December 31 2003'*.

DATE_FORMAT('2003-12-31', '%D of %M') returns *'31st of December'*.

Names of days of the week, months, etc., are always given in English, regardless of the MySQL language setting (*language* option).

STR_TO_DATE is the inverse function to *DATE_FORMAT*: *SELECT STR_TO_DATE('December 31 2005', '%M %d %Y')* returns *'2005-12-31'*.

Date Symbols in DATE_FORMAT, TIME_FORMAT, and FROM_UNIXTIME		
%W	day of week	*Monday* to *Sunday*
%a	day of week abbreviated	*Mon* to *Sun*
%e	day of month	*1* to *31*
%d	day of month two-digit	*01* to *31*
%D	day of month with ending	*1st, 2nd, 3rd, 4th, …*
%w	day of week as number	*0* (*Sunday*) to *6* (*Saturday*)
%j	day in year, three-digit	*001* to *366*
%U	week number, two-digit (Sunday)	*00* to *52*
%u	week number, two-digit (Monday)	*00* to *52*
%M	name of month	*January* to *December*
%b	name of month abbreviated	*Jan* to *Dec*
%c	month number	*1* to i
%m	month number, two-digit	*01* to *12*
%Y	year, four-digit	*2002, 2003, …*
%y	year, two-digit	*00, 01, …*
%%	the symbol %	*%*

A few remarks about the week number are in order: *%U* returns 0 for the days from before the first Sunday in the year. From the first Sunday until the following Saturday, it returns 1, then 2, etc. With *%u* you get the same thing, with the first Sunday replaced by the first Monday.

Time Symbols in DATE_FORMAT, TIME_FORMAT, and FROM_UNIXTIME		
%f	microseconds	000000 to 99999
%S or *%s*	seconds, two-digit	00 to 59
%i	minutes, two-digit	00 to 59
%k	hours (24-hour clock)	0 to 23
%H	hours, two-digit, 0 to 23 o'clock	00 to 23
%l	hours (12-hour clock)	1 to 12
%h or *%I*	hours, two-digit, to 12 o'clock	01 to 12
%T	24-hour clock	00:00:00 to 23:59:59
%r	12-hour clock	12:00:00 AM to 11:59:59 PM
%p	AM or PM	AM, PM

In order to avoid having to re-create frequently used formatting strings, since MySQL 4.1 you can get help from the function *GET_FORMAT*.

```
SELECT DATE_FORMAT(NOW(), GET_FORMAT(DATE, 'EUR'))
  31.12.2005
```

The following table summarizes the results of these functions and gives examples of the resulting formatting of dates and times.

GET_FORMAT Formatting Codes

GET_FORMAT(DATE, 'USA')	*'%m.%d.%Y'*	*12.31.2005*
GET_FORMAT(DATE, 'EUR')	*'%d.%m.%Y'*	*31.12.2005*
GET_FORMAT(DATE, 'ISO')	*'%Y-%m-%d'*	*2005-12-31*
GET_FORMAT(DATE, 'JIS')	*'%Y-%m-%d'*	*2005-12-31*
GET_FORMAT(DATE, 'INTERNAL')	*'%Y%m%d'*	*20051231*
GET_FORMAT(TIME, 'USA')	*'%h:%i:%s %p'*	*11:59:59 PM*
GET_FORMAT(TIME, 'EUR')	*'%H:%i:%s'*	*23:59:59*
GET_FORMAT(TIME, 'ISO')	*'%H:%i:%s'*	*23:59:59*
GET_FORMAT(TIME, 'JIS')	*'%H:%i:%s'*	*23:59:59*
GET_FORMAT(TIME, 'INTERNAL')	*'%H%i%s'*	*235959*
GET_FORMAT(DATETIME, 'USA')	*'%h:%i:%s %p %h:%i:%s %p'*	
GET_FORMAT(DATETIME, 'EUR')	*'%H:%i:%s %H:%i:%s'*	
GET_FORMAT(DATETIME, 'ISO')	*'%H:%i:%s %H:%i:%s'*	
GET_FORMAT(DATETIME, 'JIS')	*'%H:%i:%s %H:%i:%s'*	
GET_FORMAT(DATETIME, 'INTERNAL')	*'%H%i%s%H%i%s'*	

GROUP BY Functions

The following functions can be used in *SELECT* queries (frequently in combination with *GROUP BY*):

```
USE mylibrary
SELECT catName, COUNT(titleID) FROM titles, categories
WHERE titles.catID=categories.catID
GROUP BY catName
ORDER BY catName
catName                 COUNT(titleID)
Children's books                     3
Computer books                       5
Databases                            2
...
```

Since MySQL 4.1, the desired sort order can be specified in some aggregate functions, as in MAX(*column COLLATE collname).*

Aggregate Functions

AVG(expr)	Computes the average of *expr.*
BIT_AND(expr)	Performs a bitwise AND of *expr.*
BIT_OR(expr)	Performs a bitwise OR of *expr.*
BIT_XOR(expr)	Performs a bitwise XOR of *expr.*
COUNT(expr)	Returns the number of expressions *expr.*
COUNT(DISTINCT expr)	Returns the number of different *expr* expressions.
GROUP_CONCAT(expr)	Concatenates the strings (since MySQL 4.1). Here is the complete syntax for *ex* *[DISTINCT] expr1, expr2 … [ORDER BY column [DESC]] [SEPARATOR '…'].* *ORDER BY* sorts the strings before concatenating them. *SEPARATOR* specifies the separator character (by default a comma). Examples of *GROUP_CONCAT* can be found in Chapter 9.

Continue

Aggregate Functions *(Continued)*	
MAX(expr)	Returns the maximum of *expr*.
MIN(expr)	Returns the minimum of *expr*.
STD(expr)	Computes the standard deviation of *expr*.
STDDEV(expr)	Like *STD(expr)*.
SUM(expr)	Computes the sum of *expr*.
VARIANCE(expr)	Computes the variance of *expr* (since MySQL 4.1).

Additional Functions

Miscellaneous	
BIT_COUNT(x)	Returns the number of set bits.
COALESCE(list)	Returns the first element of the list that is not *NULL*.
LOAD_FILE(filename)	Loads a file from the local file system.

Administrative Functions	
BENCHMARK(n, expr)	Executes *expr* a total of *n* times and measures the time elapsed.
CONNECTION_ID( )	Returns the ID number of the current database connection.
CURRENT_USER( )	Returns the name of the current user in the form in which authentication takes place (that is, with the IP number instead of the host name, e.g., *"radha@127.0.0.1"*).
DATABASE( )	Returns the name of the current database.
FOUND_ROWS( )	Returns since MySQL 4.0 the number of records found by a *SELECT LIMIT* query if in the *SELECT* command, the option *SQL_CALC_FOUND_ROWS* was used.
GET_LOCK(name, time)	Defines a lock with the name *name* for the time *time* (in seconds); see also Chapter 10.
IDENTITY( )	Since MySQL 4.0 equivalent to *LAST_INSERT_ID( )*.
IS_FREE_LOCK(name)	Tests whether the lock *name* is available; returns 0 if the lock is currently in use (thus before *GET_LOCK* was executed), otherwise, 1.
LAST_INSERT_ID( )	Returns the *AUTO_INCREMENT* number most recently generated within the current connection to the database.
RELEASE_LOCK(name)	Releases the lock *name*.
SESSION_USER( )	Equivalent to *USER( )*.
SYSTEM_USER( )	Equivalent to *USER( )*.
USER( )	Returns the name of the current user and associated host name (e.g., *"root@localhost"*).
VERSION( )	Returns the MySQL version number as a string.

GIS Data Types and Functions

The GIS data types and functions described here have been available since MySQL 4.1. The GIS data types collected in the following table can be used in the declaration of a table column (such as *INT* or *VARCHAR*). At present, these data types can be used only in MyISAM tables, and not in InnoDB tables. Optionally, geometric columns can be equipped with a *SPATIAL INDEX*, which is independent of the GIS data type used.

GIS Data Types	
GEOMETRY	Can accept any of the following data types.
POINT	Stores a single point. Example: *POINT(10 10)*.
MULTIPOINT	Stores a list of points. Example: *MULTIPOINT(10 10, 0 20, -3 2)*.
LINESTRING	Stores a line segment. Example: *LINESTRING(0 0, 1 1, 3 3)*.
MULTILINESTRING	Stores several line strings. Example: *MULTILINESTRING((0 0, 5 5), (10 10, 20 20, 40 20), (10 10, 2 0))*.
POLYGON	Stores a closed polygon, which is permitted to have holes. Example: *POLYGON((1 1, 9 1, 9 9, 1 9, 1 1), (3 3, 3 6, 6 6, 6 3, 3 3))*.
MULTIPOLYGON	Stores several polygons. Example: *MULTIPOLYGON(((0 0, 5 0, 5 5, 0 5, 0 0)), ((10, 10, 30 30, 30 10, 10 10)))*.
GEOMETRYCOLLECTION	Stores a list of arbitrary geometric objects. Example: *GEOMETRYCOLLECTION(POINT(100 100), POINT(10 10), LINESTRING (1 1, 100 1, 100 100))*.

Conversion Functions	
ASTEXT(geom)	Returns a geometric object as *well-known text*.
ASBINARY(geom)	Returns a geometric object as *well-known binary*.
GEOMFROMTEXT(txt [, srid])	Creates the MySQL-internal geometric format out of a *well-known text* (WKT) string. Optionally, an identifier for the coordinate system can be given, which is stored by MySQL but otherwise ignored.
GEOMFROMWKB(bindata)	Creates the MySQL-internal geometry format from binary data in WKB format.

General Geometric Functions (for All GIS Data Types)	
DIMENSION(g)	Returns the dimension of the object. Possible results are: 1 for empty objects, 0 for points (length = 0, area = 0), 1 for lines (length > 0, area = 0), 2 for polygons, etc. (length > 0, area > 0).
ENVELOPE(g)	Returns the bounding box of the geometric object (see below). The result has the data type *POLYGON*.
GEOMETRYTYPE(g)	Returns the type of the geometric object as a string: (*POINT*, *LINESTRING*, *POLYGON*, …).
SRID(g)	Returns the identifier of the coordinate system. (MySQL stores a coordinate system identifier for all geometric objects, but it does evaluate this information.)

POINT Functions	
X(pt)	Returns the X coordinate.
Y(pt)	Returns the Y coordinate.

LINESTRING Functions (in Part Also MULTILINESTRING)	
GLENGTH(ls)	Returns the length of the line as a floating-point number.
ISCLOSED(ls)	Returns 1 if the starting point is equal to the endpoint, otherwise 0.
NUMPOINTS(ls)	Returns the number of points that the line consists of.
STARTPOINT(ls)	Returns the first point (*POINT* object).
ENDPOINT(ls)	Returns the last point.
POINTN(ls, n)	Returns the *n*th point.

POLYGON Functions (in Part Also *MULTIPOLYGON*)	
AREA(p)	Returns the area of the polygon as a floating-point number.
EXTERIORRING(p)	Returns the exterior ring of the polygon as a *LINESTRING* object.
INTERIORRINGN(p, n)	Returns the interior ring of the polygon as a *LINESTRING* object.
NUMINTERIORRINGS(p)	Returns the number of interior rings (holes).

GEOMETRYCOLLECTION Functions	
GEOMETRYN(gc, n)	Returns the geometric object at location *n*.
NUMGEOMETRIES(gc)	Returns the number of objects in a collection.

The OpenGIS specification provides for the following analysis functions in two variants. The fast variant takes into account only the bounding box of the geometric object (*MBRname*, where *MBR* stands for *minimum bounding rectangle*). The precise variant, on the other hand, calculates with the complete geometric data (*name*). MySQL formally recognizes both variants, but internally they are identical, and only the bounding box is considered.

Analysis Functions for GIS Data Types	
[MBR]CONTAINS(g1, g2)	Returns *TRUE*, if *g2* is completely contained in *g1*.
[MBR]WITHIN(g1, g2)	Returns *TRUE*, if *g1* is completely contained within *g2*. (*CONTAINS(a, b)* is equivalent to *WITHIN(b, a)*.)
[MBR]EQUAL(g1, g2)	Returns *TRUE* if *g1* and *g2* are equal.
[MBR]INTERSECTS(g1, g2)	Returns *TRUE* if *g1* and *g2* intersect.
[MBR]OVERLAPS(g1, g2)	Returns *TRUE* if *g1* and *g2* overlap.
[MBR]TOUCHES(g1, g2)	Returns *TRUE* if *g1* and *g2* touch each other.
[MBR]DISJOINT(g1, g2)	Returns *TRUE* if *g1* and *g2* neither overlap nor touch.

Language Elements for Stored Procedures and Triggers

You can place almost any SQL command or function described in this chapter in the code of a stored procedure or trigger. Furthermore, MySQL provides some additional language elements with which you can declare system variables and cursors, form loops and queries, and so on. The syntax of these language elements is summarized in the following tables.

Summary of Commands, Loops, and Queries	
[blockname:] BEGIN DECLARE variables, cursors, conditions, handlers etc.; commands …; END [blockname];	*BEGIN* introduces a block of commands; *END* ends the block. First, variables, cursors, etc., can be declared within a block, followed by the SQL commands and additional language elements.
LEAVE blockname;	If the block has a name (*blockname*), then it can be exited early with *LEAVE*. Code execution is continued after *END blockname*.
RETURN result;	With functions, a result can be returned with *RETURN*. At the same time, code execution ends.

Queries	
IF condition1 THEN commands ...; [ELSE IF condition2 THEN commands ...;] [ELSE commands ...;] END IF;	*IF/END IF* creates a simple branch. Arbitrarily many *ELSE-IF* clauses are allowed, but only one terminating *ELSE* branch.
CASE expression WHEN value1 THEN commands ...; [ELSE commands ...;] END CASE;	*CASE/END CASE* helps to formulate case decisions with several variants in a more understandable manner. Arbitrarily many *WHEN* branches are allowed. (Each *CASE* can be formulated equivalently with *IF/END IF*.)

Loops	
[loopname:] REPEAT commands ...; UNTIL condition END REPEAT [loopname];	*REPEAT* loops are executed until the condition is satisfied (at least once).
[loopname:] WHILE condition DO commands ...; END WHILE [lpname];	*WHILE* loops are executed while the condition is satisfied. If the condition is *FALSE* (0) before the first execution of the loop, the loop is not executed at all.
loopname: LOOP commands ...; END LOOP loopname;	*LOOP/END LOOP* creates an infinite loop. It must be ended with *BREAK*.
LEAVE loopname;	Ends a loop prematurely.
ITERATE loopname;	Repeats the commands of the loop body one more time.

The following *DECLARE* instructions must be executed at the beginning of a code block and in the order given (that is, first variable declaration, then cursors, then conditions and handlers).

Variables, Cursors , Conditions, Handlers (DECLARE)	
DECLARE varname1, varname2, … datatype [DEFAULT value];	Declares the local variables *varname1*, *varname2*, etc. with a particular data type (e.g., *INT*). Optionally, a default value can be assigned. The variables can be used only within the block in which they are defined. Their names cannot coincide with those of a table or column.
DECLARE cursorname CURSOR FOR select-command;	Declares a cursor for step-by-step processing of the results of a *SELECT* command.
DECLARE name CONDITION FOR condition1, c2, c3 …;	Names one or more error conditions. There are several possibilities for specifying the error conditions: *SQLSTATE 'code'* *n* (MySQL error number) *SQLWARNING* *NOT FOUND* *SQLEXCEPTION*
DECLARE CONTINUE | EXIT HANDLER FOR condition1, c2, c3 … command;	Declares a handler for error-handling. With *CONTINUE* the code is simply continued, while with *EXIT* the current code block is exited. First, any *command* is executed. In addition to the above variants, for the error conditions a *CONDITION* is allowed.

The following table shows the application of a cursor.

Cursor Application	
DECLARE done INT DEFAULT 0;	Defines the variable for the handler.
DECLARE var1, var2 … datatype;	Declares the variables for the *SELECT* command.
DECLARE mycursor CURSOR FOR select command;	Declares the cursor.
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=1;	Declares a handler that becomes active after the last record is read.
OPEN mycursor;	Activates the cursor.
myloop: LOOP	
FETCH mycursor INTO var1, var2 …; *IF done=1 THEN LEAVE myloop; END IF;*	
END LOOP myloop;	Within the loop, reads the records of the *SELECT* command into the variables *var1*, *var2*, etc. If there are no further records, the handler assigns the value 1, and the loop is ended.
CLOSE mycursor;	Closes the cursor.