16

SQL Syntax for MySQL

In this chapter, we cover the full range of SQL supported by MySQL. If you are interested in compatibility with other SQL databases, MySQL supports the ANSI SQL2 standard. In that case, you should avoid using any proprietary MySQL extensions to the SQL standard.

Basic Syntax

SQL is a kind of controlled English language consisting of verb phrases. These verb phrases begin with a SQL command followed by other SQL keywords, literals, identfiers, or punctuation. Keywords are never case sensitive. Identifiers for database names and table names are case sensitive when the underlying file system is case sensitive (all UNIX except Mac OS X) and case insensitive when the underlying file system is case insensitive (Mac OS X and Windows). You should, however, avoid referring to the same database or table name in a single SQL statement using different cases—even if the underlying operating system is case insensitive. For example, the following SQL is troublesome:

```
SELECT TBL.COL FROM tbl;
```

Table aliases are case sensitive, but column aliases are case insensitive.

If all of this case sensitivity nonsense is annoying to you, you can force MySQL to convert all table names to lower case by starting *mysqld* with the argument *-O lower_case_table_names=1*.

Literals

Literals come in the following varieties:

String Literals

String literals may be enclosed either by single quotes or double quotes. If you wish to be ANSI compatible, you should always use single quotes. Within a string literal, you may represent special characters through escape sequences. An escape sequence is a backslash followed by another character to indicate to MySQL that the second character has a meaning other than its normal meaning. Table 16-1 shows the MySQL escape sequences. Quotes can also be escaped by doubling them up: `'This is a ''quote'''`. However, you do not need to double up on single quotes when the string is enclosed by double quotes: `"This is a 'quote'"`.

Binary Literals

Like string literals, binary literals are enclosed in single or double quotes. You must use escape sequences in binary data to escape `NUL` (ASCII 0), `"` (ASCII 34), `'` (ASCII 39), and \ (ASCII 92).

Number Literals

Numbers appear as a sequence of digits. Negative numbers are preceded by a `-` sign and a `.` indicates a decimal point. You may also use scientific notation: -45198.2164e+10.

Hexadecimal Literals

MySQL also supports the use of hexadecimal literals in SQL. The way in which that hexadecimal is interpreted is dependent on the context. In a numeric context, the hexadecimal literal is treated is a numeric value. Absent of a numeric context, it is treated as a binary value. This 0x1 + 1 is 2, but 0x4d7953514c by itself is 'MySQL'.

Null

The special keyword `NULL` signifies a null literal in SQL. In the context of import files, the special escape sequence `\N` signifies a null value.

Table 16-1. . MySQL Escape Sequences

Escape Sequence	Value
`\0`	NUL
`\'`	Single quote
`\"`	Double quote
`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\z`	Ctrl-z (workaround for Windows use of ctrl-z as EOF)
`\\`	Backslash

Table 16-1. . MySQL Escape Sequences

Escape Sequence	Value
\%	Percent sign (only in contexts where a percent sign would be interpreted as a wild card)
_	Underscore (only in contexts where an underscore would be interpreted as a wild card)

Identifiers

Identifiers are names you make up to reference database objects. In MySQL, database objects consist of databases, tables, and columns. These objects fit into a hierarchical namespace whose root element is the database in question. You can reference any given object on a MySQL server—assuming you have the proper rights—in one of the following conventions:

Absolute Naming

Absolute naming is specifying the full tree of the object you are referencing. For example, the column `BALANCE` in the table `ACCOUNT` in the database `BANK` would be referenced absolutely as:
`BANK.ACCOUNT.BALANCE`

Relative Naming

Relative naming allows you to specify only part of the object's name with the rest of the name being assumed based on your current context. For example, if you are currently connected to the `BANK` database, you can reference the `BANK.ACCOUNT.BALANCE` column simply as `ACCOUNT.BALANCE`. In a SQL query where you have specified you are selecting from the `ACCOUNT` table, you can reference the column using only `BALANCE`. You must provide an extra layer of context whenever relative naming might result in ambiguity. An example of such ambiguity would be a `SELECT` statement pulling from two tables that both have `BALANCE` columns.

Aliasing

Aliasing enables you to reference an object using an alternate name that helps avoid both ambiguity and the need to fully qualify a long name.

In general, MySQL allows you to use any character in an identifier.* This rule is limited, however, for databases and tables since these values must be treated as files on the local file system. You can therefore use only characters valid for the underlying file system's file naming conventions in a database or table name. Spe-

* Older versions of MySQL limited identifiers to valid alphanumeric characters from the default character set as well as $ and _.

cifically, you may not use / or . in a database or table name. You can never use NUL (ASCII 0) or ASCII 255 in an identifier.

Given these rules, it is very easy to shoot yourself in the foot when naming things. As a general rule, it is a good idea to stick to alphanumeric characters from whatever character set you are using.

When an identifier is also a SQL keyword, you must enclose the identifier in backticks:

```
CREATE TABLE `select` ( `table` INT NOT NULL PRIMARY KEY AUTO_INCREMENT);
```

Since MySQL 3.23.6, MySQL supports the quoting of identifiers using both backticks and double quotes. For ANSI compatibility, however, you should use double quotes for quoting identifiers. You must, however, be running MySQL in ANSI mode.

Comments

You can introduce comments in your SQL to specify text that should not be interpreted by MySQL. This is particularly useful in batch scripts for creating tables and loading data. MySQL specifically supports three kinds of commenting: C, shell-script, and ANSI SQL commenting.

C commenting treats anything between /* and */ as comments. Using this form of commenting, your comments can span multiple lines. For example:

```
/*
 * Creates a table for storing customer account information.
*/
DROP TABLE IF EXISTS ACCOUNT;

CREATE TABLE ACCOUNT ( ACCOUNT_ID BIGINT NOT NULL PRIMARY KEY AUTO_INCREMENT,
                       BALANCE DECIMAL(9,2) NOT NULL );
```

Within C comments, MySQL still treats single quotes and double quotes as a start to a string literal. In addition, a semi-colon in the comment will cause MySQL to think you are done with the current statement.

Shell-script commenting treats anything from a # character to the end of a line as a comment:

```
CREATE TABLE ACCOUNT ( ACCOUNT_ID BIGINT NOT NULL PRIMARY KEY AUTO_INCREMENT,
                       BALANCE DECIMAL(9,2) NOT NULL ); # Not null ok?
```

MySQL does not really support ANSI SQL commenting, but it comes close. ANSI SQL commenting is -- to the end of a line. MySQL supports two dashes and a space ('-- ') followed by the comment. The space is the non-ANSI part:

```
DROP TABLE IF EXISTS ACCOUNT; -- Drop the table if it already exists
```

SQL Commands

ALTER TABLE

Syntax

```
ALTER [IGNORE] TABLE table action_list
```

Description

The ALTER statement covers a wide range of actions that modify the structure of a table. This statement is used to add, change, or remove columns from an existing table as well as to remove indexes. To perform modifications on the table, MySQL creates a copy of the table and changes it, meanwhile queuing all table altering queries. When the change is done, the old table is removed and the new table put it its place. At this point the queued queries are performed. As a safety precaution, if any of the queued queries create duplicate keys that should be unique, the ALTER statement is rolled back and cancelled. If the IGNORE keyword is present in the statement, duplicate unique keys are ignored and the ALTER statement proceeds as if normal. Be warned that using IGNORE on an active table with unique keys is inviting table corruption. Possible actions include:

ADD [COLUMN] *create_clause* [FIRST | AFTER *column*]
: Adds a new column to the table. The *create_clause* is simply the SQL that would define the column in a normal table creation. The column will be created as the first column if the FIRST keyword is specified. Alternately, you can use the AFTER keyword to specify which column it should be added after. If neither FIRST nor AFTER is specified, then the column is added at the end of the table's column list. You may add multiple columns at once by separating create clauses by commas.

ADD INDEX [*name*] (*column*, ...)
: Adds an index to the altered table. If the name is omitted, one will be chosen automatically by MySQL.

ADD PRIMARY KEY (*column*, ...)
: Adds a primary key consisting of the specified columns to the table. An error occurs if the table already has a primary key.

ADD UNIQUE[*name*] (*column*, ...)
: Adds a unique index to the altered table similar to the ADD INDEX statement.

`ALTER [COLUMN] column SET DEFAULT value`

Assigns a new default value for the specified column. The `COLUMN` keyword is optional and has no effect.

`ALTER [COLUMN] column DROP DEFAULT`

Drops the current default value for the specified column. A new default value will be assigned to the column based on the `CREATE` statement used to create the table. The `COLUMN` keyword is optional and has no effect.

`CHANGE [COLUMN] column create_clause`
`MODIFY [COLUMN] create_clause`

Alters the definition of a column. This statement is used to change a column from one type to a different type while affecting the data as little as possible. The create clause is a full clause as specified in the `CREATE` statement. This includes the name of the column. The `MODIFY` version is the same as `CHANGE` if the new column has the same name as the old. The `COLUMN` keyword is optional and has no effect. MySQL will try its best to perform a reasonable conversion. Under no circumstance will MySQL give up and return an error when using this statement; a conversion of some sort will always be done. With this in mind you should (1) make a backup of the data before the conversion and (2) immediately check the new values to see if they are reasonable.

DROP [COLUMN] *column*

Deletes a column from a table. This statement will remove a column and all of its data from a table permanently. There is no way to recover data destroyed in this manner other than from backups. All references to this column in indices will be removed. Any indices where this was the sole column will be destroyed as well. (The `COLUMN` keyword is optional and has no effect.)

DROP PRIMARY KEY

Drops the primary key from the table. If no primary key is found in the table, the first unique key is deleted.

DROP INDEX *key*

Removes an index from a table. This statement will completely erase an index from a table. This statement will not delete or alter any of the table data itself, only the index data. Therefore, an index removed in this manner can be recreated using the `ALTER TABLE ... ADD INDEX` statement.

RENAME [AS] *new_table*
RENAME [TO] *new_table*

Changes the name of the table. This operation does not affect any of the data or indices within the table, only the table's name. If this statement is performed alone, without any other ALTER TABLE clauses, MySQL will not create a temporary table as with the other clauses, but simply perform a fast Unix-level rename of the table files.

ORDER BY *column*

Forces the table to be re-ordered by sorting on the specified column name. The table will no longer be in this order when new rows are inserted. This option is useful for optimizing tables for common sorting queries.

table_options

Enables a redefinition of the tables options such as the table type.

Multiple ALTER statements may be combined into one using commas as in the following example:

```
ALTER TABLE mytable DROP myoldcolumn, ADD mynewcolumn INT
```

MySQL also provides support for actions to alter the FOREIGN KEY, but they do nothing. . The syntax is there simply for compatibility with other databases.

To perform any of the ALTER TABLE actions, you must have SELECT, INSERT, DELETE, UPDATE, CREATE, and DROP privileges for the table in question.

Examples

```
# Add the field 'address2' to the table 'people' and make
# it of type 'VARCHAR' with a maximum length of 200.
ALTER TABLE people ADD COLUMN address2 VARCHAR(100)
# Add two new indexes to the 'hr' table, one regular index for the
# 'salary' field and one unique index for the 'id' field. Also, continue
# operation if duplicate values are found while creating
# the 'id_idx' index (very dangerous!).
ALTER TABLE hr ADD INDEX salary_idx ( salary )
ALTER IGNORE TABLE hr ADD UNIQUE id_idx ( id )
# Change the default value of the 'price' field in the
# 'sprockets' table to $19.95.
ALTER TABLE sprockets ALTER price SET DEFAULT '$19.95'
# Remove the default value of the 'middle_name' field in the 'names' table.
ALTER TABLE names ALTER middle_name DROP DEFAULT
# Change the type of the field 'profits' from its previous value (which was
# perhaps INTEGER) to BIGINT.
ALTER TABLE finanaces CHANGE COLUMN profits profits BIGINT
# Remove the 'secret_stuff' field from the table 'not_private_anymore'
ALTER TABLE not_private_anymore DROP secret_stuff
# Delete the named index 'id_index' as well as the primary key from the
# table 'cars'.
ALTER TABLE cars DROP INDEX id_index, DROP PRIMARY KEY
# Rename the table 'rates_current' to 'rates_1997'
ALTER TABLE rates_current RENAME AS rates_1997
```

CREATE DATABASE

Syntax

```
CREATE DATABASE dbname
```

Description

Creates a new database with the specified name. You must have the proper privileges to create the database. Running this command is the same as running the *mysqladmin create* utility.

Example

```
CREATE DATABASE Bank;
```

CREATE FUNCTION

Syntax

```
CREATE [AGGREGATE] FUNCTION name
RETURNS return_type SONAME library
```

Description

The CREATE FUNCTION statement allows MySQL statements to access precompiled executable functions. These functions can perform practically any operation, since they are designed and implemented by the user. The return value of the function can be STRING, for character data; REAL, for floating point numbers; or INTEGER for integer numbers. MySQL will translate the return value of the C function to the indicated type. The library file that contains the function must be a standard shared library that MySQL can dynamically link into the server.

Example

```
CREATE FUNCTION multiply RETURNS REAL SONAME mymath
```

CREATE INDEX

Syntax

```
CREATE [UNIQUE] INDEX name ON table (column, ...)
```

Description

The CREATE INDEX statement is provided for compatibility with other implementations of SQL. In older versions of SQL this statement does nothing. As of 3.22, this statement is equivalent to the ALTER TABLE ADD INDEX statement. To per-

form the CREATE INDEX statement, you must have INDEX privileges for the table in question.

Example

```
CREATE UNIQUE INDEX TransIDX ON Translation ( language, locale, code );
```

CREATE TABLE

Syntax

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] table
(create_clause, ...) [table_options]
[[IGNORE|REPLACE] select]
```

Description

The CREATE TABLE statement defines the structure of a table within the database. This statement is how all MySQL tables are created. If the TEMPORARY keyword is used, the table exists only as long as the current client connection exists, unless it is dropped first

The IF NOT EXISTS clause tells MySQL to create the table only if the table does not already exist. If the table does exist, nothing happens. If the table exists and IF NOT EXISTS and TEMPORARY are not specified, an error will occur. If TEMPORARY is specified and the table exists but IF NOT EXISTS is not specified, the existing table will simply be invisible to this client for the duration of the new temporary table's life.

This statement consists of the name of the new table followed by any number of field definitions. The syntax of a field definition is the name of the field followed by its type, followed by any modifiers (e.g., name char(30) not null). MySQL supports the data types described in Chapter 17. The allowed modifiers are:

DEFAULT *value*
: This attribute assigns a default value to a field. If a row is inserted into the table without a value for this field, this value will be inserted. If a default is not defined, a null value is inserted unless the field is defined as NOT NULL in which case MySQL picks a value based on the type of the field.

NOT NULL
: This attribute guarantees that every entry in the column will have some non-NULL value. Attempting to insert a NULL value into a field defined with NOT NULL will generate an error.

NULL
: This attribute specifies that the field is allowed to contain NULL values. This is the default if neither this nor the NOT NULL modifier are specified. Fields that

are contained within an index cannot contain the NULL modifier. (It will be ignored, without warning, if it does exist in such a field.)

`PRIMARY KEY`

This attribute automatically makes the field the primary key (see later) for the table. Only one primary key may exist for a table. Any field that is a primary key must also contain the NOT NULL modifier.

`REFERENCES` *`table`* `[(`*`column`*`, . . .)] [MATCH FULL | MATCH PARTIAL] [ON DELETE` *`option`*`] [ON UPDATE` *`option`*`]`

This attribute currently has no effect. MySQL understands the full references syntax but does not implement its behavior. The modifier is included to make it easier to import SQL from different SQL sources. In addition, this functionality may be included in a future release of MySQL.

MySQL supports the concept of an index of a table, as described in Chapter 8, *Database Design*. Indexes are created by means of special "types" that are included with the table definition:

`FULLTEXT (` *`column`*`, ... )`

Since MySQL 3.23.23, MySQL has supported full text indexing. To create a full text index, use the `FULLTEXT` keyword:

```
CREATE TABLE Item ( itemid INT NOT NULL PRIMARY KEY,
                    name VARCHAR(25) NOT NULL,
                    description TEXT NOT NULL,
                    FULLTEXT ( name, description )
);
```

`KEY/INDEX [`*`name`*`] (`*`column`*`, ...)`

Creates a regular index of all of the named columns (`KEY` and `INDEX`, in this context, are synonyms). Optionally the index may be given a name. If no name is provided, a name is assigned based on the first column given and a trailing number, if necessary, for uniqueness. If a key contains more than one column, leftmost subsets of those columns are also included in the index. Consider the following index definition.

```
INDEX idx1 ( name, rank, serial );
```

When this index is created, the following groups of columns will be indexed:

— `name, rank, serial`

— `name, rank`

— `name`

`PRIMARY KEY`

Creates the primary key of the table. A primary key is a special key that can be defined only once in a table. The primary key is a `UNIQUE` key with the

name "PRIMARY." Despite it's privileged status, in function it is the same as every other unique key.

`UNIQUE [`*`name`*`] (`*`column`*`, ...)`

Creates a special index where every value contained in the index (and therefore in the fields indexed) must be unique. Attempting to insert a value that already exists into a unique index will generate an error. The following would create a unique index of the "nicknames" field:

```
UNIQUE (nicknames);
```

when indexing character fields (CHAR, VARCHAR and their synonyms only), it is possible to index only a prefix of the entire field. For example, this following will create an index of the numeric field 'id' along with the first 20 characters of the character field 'address':

```
INDEX adds ( id, address(20) );
```

When performing any searches of the field 'address', only the first 20 characters will be used for comparison unless more than one match is found that contains the same first 20 characters, in which case a regular search of the data is performed. Therefore, it can be a big performance bonus to index only the number of characters in a text field that you know will make the value unique.

Fields contained in an index must be defined with the `NOT NULL` modifier. When adding an index as a separate declaration, MySQL will generate an error if `NOT NULL` is missing. However, when defining the primary key by adding the `PRIMARY KEY` modifier to the field definition, the `NOT NULL` modifier is automatically added (without a warning) if it is not explicitly defined.

In addition to the above, MySQL supports the following special "types":

- FOREIGN KEY *(name (column, [column2, . . .])*
- CHECK

These keywords do not actually perform any action. They exist so that SQL exported from other databases can be more easily read into MySQL. Also, some of this missing functionality may be added into a future version of MySQL.

As of MySQL 3.23, you can specify table options at the end of a `CREATE TABLE` statement. These options are:

`AUTO_INCREMENT =` *`start`*

Specifies the first value to be used for an `AUTO_INCREMENT` column.

`AVG_ROW_LENGTH =` *`length`*

An option for tables containing large amounts of variable-length data. The average row length is an optimization hint to help MySQL manage this data.

`CHECKSUM = 0 or 1`

When set to 1, this option forces MySQL to maintain a checksum for the table to improve data consistency. This option creates a performance penalty.

`COMMENT = comment`

Provides a comment for the table. The comment may not exceed 60 characters.

`DELAY_KEY_WRITE = 0 or 1`

For MyISAM tables only. When set, this option delays key table updates until the table is closed.

`MAX_ROWS = rowcount`

The maximum number of rows you intend to store in the table.

`MIN_ROWS = rowcount`

The minimum number of rows you intend to store in the table.

`PACK_KEYS = 0 or 1`

For MyISAM and ISAM tables only. This option provides a performance booster for heavy-read tables. Set to 1, this option causes smaller keys to be created and thus slows down writes while speeding up reads.

`PASSWORD = 'password'`

Only available to MySQL customers with special commercial licenses. This option uses the specified password to encrypt the table's *.frm* file. This option has no effect on the standard version of MySQL.

`ROW_FORMAT = DYNAMIC or STATIC`

For MyISAM tables only. Defines how the rows should be stored in a table.

`TYPE = rowtype`

Specifies the table type of the database. If the selected table type is not available, then the closest table type available is used. For example, BDB is not available yet for Mac OS X. If you specified `TYPE=BDB` on a Mac OS X system, MySQL will instead create the table as a MyISAM table. Table 16-2 contains a list of supported table types and their advanatages. For a more complete discussion of MySQL tables types, see the MySQL table type reference.

Table 16-2. . MySQL Table Types

Type	Transactional	Description
BDB	yes	Transaction-safe tables with page locking.
Berkeley_db	yes	Alias for BDB
HEAP	no	Memory-based table. Not persistent.
ISAM	no	Ancient format. Replaced by MyISAM.
InnoDB	yes	Transaction-safe tables with row locking.
MERGE	no	A collection of MyISAM tables merged as a single table.

Table 16-2. . MySQL Table Types

Type	Transactional	Description
MyISAM	no	A newer table type to replace ISAM that is portable.

You must have CREATE privileges on a database to use the CREATE TABLE statement.

Examples

```
# Create the new empty database 'employees'
CREATE DATABASE employees;
# Create a simple table
CREATE TABLE emp_data ( id INT, name CHAR(50) );
# Make the function make_coffee (which returns a string value and is stored
# in the myfuncs.so shared library) available to MySQL.
CREATE FUNCTION make_coffee RETURNS string SONAME "myfuncs.so";
```

DELETE

Syntax

```
DELETE [LOW_PRIORITY] FROM table [WHERE clause] [LIMIT n]
```

Description

Deletes rows from a table. When used without a WHERE clause, this will erase the entire table and recreate it as an empty table. With a clause, it will delete the rows that match the condition of the clause. This statement returns the number of rows deleted to the user.

As mentioned above, not including a WHERE clause will erase this entire table. This is done using an efficient method that is much faster than deleting each row individually. When using this method, MySQL returns 0 to the user because it has no way of knowing how many rows it deleted. In the current design, this method simply deletes all of the files associated with the table except for the file that contains the actual table definition. Therefore, this is a handy method of zeroing out tables with unrecoverably corrupt data files. You will lose the data, but the table structure will still be in place. If you really wish to get a full count of all deleted tables, use a WHERE clause with an expression that always evaluates to true:

```
DELETE FROM TBL WHERE 1 = 1;
```

The LOW_PRIORITY modifier causes MySQL to wait until no clients are reading from the table before executing the delete.

The LIMIT clauses establishes the maximum number of rows that will be deleted in a single shot.

You must have DELETE privileges on a database to use the DELETE statement.

Examples

```
# Erase all of the data (but not the table itself) for the table 'olddata'.
DELETE FROM olddata
# Erase all records in the 'sales' table where the 'syear' field is '1995'.
DELETE FROM sales WHERE syear=1995
```

DESCRIBE

Syntax

```
DESCRIBE table [column]
DESC table [column]
```

Description

Gives information about a table or column. While this statement works as advertised, its functionality is available (along with much more) in the SHOW statement. This statement is included solely for compatibility with Oracle SQL. The optional column name can contain SQL wildcards, in which case information will be displayed for all matching columns.

Example

```
# Describe the layout of the table 'messy'
DESCRIBE messy
# Show the information about any columns starting
# with 'my_' in the 'big' table.
# Remember: '_' is a wildcard, too, so it must be
# escaped to be used literally.
DESC big my\_%
```

DESC

Synonym for DESCRIBE.

DROP DATABASE

Syntax

```
DROP DATABASE [IF EXISTS] name
```

Description

Permanently remove a database from the MySQL. Once you execute this statement, none of the tables or data that made up the database are available. All of the support files for the database are deleted from the file system. The number of files deleted will be returned to the user. Because three files represent most tables, the number returned is usually the number of tables times three. This is equivalent to running the *mysqladmin drop* utility. As with running *mysqladmin*, you must be the administrative user for MySQL (usually root or mysql) to perform this state-

ment.You may use the IF EXISTS clause to prevent any error message that would result from an attempt to drop a non-existent table.

DROP FUNCTION

Syntax

```
DROP FUNCTION name
```

Description

Will remove a user defined function from the running MySQL server process. This does not actually delete the library file containing the function. You may add the function again at any time using the CREATE FUNCTION statement. In the current implementation DROP FUNCTION simply removes the function from the function table within the MySQL database. This table keeps track of all active functions.

DROP INDEX

Syntax

```
DROP INDEX idx_name ON tbl_name
```

Description

Provides for compatibility with other SQL implementations. In older versions of MySQL, this statement does nothing. As of 3.22, this statement is equivalent to ALTER TABLE ... DROP INDEX. To perform the DROP INDEX statement, you must have SELECT, INSERT, DELETE, UPDATE, CREATE, and DROP privileges for the table in question.

DROP TABLE

Syntax

```
DROP TABLE [IF EXISTS] name [, name2, ...]
```

Description

Will erase an entire table permanently. In the current implementation, MySQL simply deletes the files associated with the table. As of 3.22, you may specify IF EXISTS to make MySQL not return an error if you attempt to remove a table that does not exist. You must have DELETE privileges on the table to use this statement.

DROP is by far the most dangerous SQL statement. If you have drop privileges, you may permanently erase a table or even an entire database. This is done without warning or confirmation. The only way to undo a DROP is to restore the table or database from backups. The lessons to be learned here are: (1) always keep backups; (2) don't use DROP unless you are really sure; and (3) always keep backups.

EXPLAIN

Syntax

```
EXPLAIN [table_name | sql_statement]
```

Description

Used with a table name, this command is an alias for `SHOW COLUMNS FROM table_name`.

Used with a SQL statement, this command displays verbose information about the order and structure of a SELECT statement. This can be used to see where keys are not being used efficiently. This information is returned as a result set with the following columns:

table
: The name of the table referenced by the result set row explaining the query.

type
: The type of join that will be performed. These types, in order of performance, are:

  system
  : A special case of the const type, this join supports a table with a single row.

  const
  : Used for tables with at most a single matching row that will be read at the start of the query. MySQL treats this value as a constant to speed up processing.

  eq_ref
  : Reads one row from the table for each combination of rows from the previous tables. It is used when all parts of the index are used by the join and the index is unique or a primary key.

ref
: Reads all rows with matching index values from the table for each combination of rows from the previous tables. This join occurs when only the leftmost part of an index or if the index is not unique or a primary key. This is a good join when the key in use matches only a few rows.

range
: Reads only the rows in a given range using an index to select the rows. They key column indicates the `key` in use and the `key_len` column contains the longest part of the key. The ref column will be `NULL` for this type.

index
: Reads all rows based on an index tree scan.

ALL
: A full table scan is done for each combination of rows from the previous tables. This join is generally a very bad thing. If you see it, you probably need to build a different SQL query or better organize your indices.

possible_keys
: Indicates which indices MySQL could use to build the join. If this column is empty, there are no relevant indices and you probably should build some to enhance performance.

key
: Indicates which index MySQL decided to use.

key_len
: Provides the length of the key MySQL decided to use for the join.

ref
: Describes which columns or constants were used with the key to build the join.

rows
: Indicates the number of rows MySQL estimates it will need to examine to perform the query.

Extra
: Additional information indicating how MySQL will perform the query.

Example

```
EXPLAIN SELECT customer.name, product.name FROM customer, product, purchases
WHERE purchases.customer=customer.id AND purchases.product=product.id
```

FLUSH

Syntax

```
FLUSH option[, option...]
```

Description

Flushes or resets various internal processes depending on the options given. You must have reload privileges to execute this statement. The option can be any of the following:

HOSTS
: Empties the cache table that stores hostname information for clients. This should be used if a client changes IP addresses, or if there are errors related to connecting to the host.

LOGS
: Closes all of the standard log files and reopens them. This can be used if a log file has changed inode number. If no specific extension has been given to the update log, a new update log will be opened with the extension incremented by one.

PRIVILEGES
: Reloads all of the internal MySQL permissions grant tables. This must be run for any changes to the tables to take effect.

STATUS
: Resets the status variables that keep track of the current state of the server.

TABLES
: Closes all currently opened tables and flushes any cached data to disk.

GRANT

Syntax

```
GRANT privilege
[ (column, ...) ] [, privilege [( column, ...) ] ...]
ON {table} TO user [IDENTIFIED BY 'password']
[, user [IDENTIFIED BY 'password'] ...]
[WITH GRANT OPTION]
```

Previous to MySQL 3.22.11, the GRANT statement was recognized but did nothing. In current versions, GRANT is functional. This statement will enable access rights to a user (or users). Access can be granted per database, table or individual column. The table can be given as a table within the current database, '*' to affect all tables within the current database, '*.*' to affect all tables within all databases or 'database.*' to effect all tables within the given database.

The following privileges are currently supported:

`ALL PRIVILEDGES/ALL`
: Effects all privileges

`ALTER`
: Altering the structure of tables

`CREATE`
: Creating new tables

`DELETE`
: Deleting rows from tables

`DROP`
: Deleting entire tables

`FILE`
: Creating and removing entire databases as well as managing log files

`INDEX`
: Creating and deleting indices from tables

`INSERT`
: Inserting data into tables

`PROCESS`
: Killing process threads

`REFERENCES`
: Not implemented (yet)

`RELOAD`
: Refreshing various internal tables (see the `FLUSH` statement)

`SELECT`
: Reading data from tables

`SHUTDOWN`
: Shutting down the database server

`UPDATE`
: Altering rows within tables

`USAGE`
: No privileges at all

The user variable is of the form *user@hostname*. Either the user or the hostname can contain SQL wildcards. If wildcards are used, either the whole name must be quoted, or just the part(s) with the wildcards (e.g., joe@"%.com " and "*joe@%.com*" are both valid). A user without a hostname is considered to be the same as *user@"%"*.

If you have a global GRANT privilege, you may specify an optional INDENTIFIED BY modifier. If the user in the statement does not exist, it will be created with the given password. Otherwise the existing user will have his or her password changed.

Giving the GRANT privilege to a user is done with the WITH GRANT OPTION modifier. If this is used, the user may grant any privilege they have onto another user.

Examples

```
# Give full access to joe@carthage for the Account table
GRANT ALL ON bankdb.Account TO joe@carthage;
# Give full access to jane@carthage for the
# Account table and create a user ID for her
GRANT ALL ON bankdb.Account TO jane@carthage IDENTIFIED BY 'mypass';
# Give joe on the local machine the ability
# to SELECT from any table on the webdb database
GRANT SELECT ON webdb.* TO joe;
```

INSERT

Syntax

```
INSERT [DELAYED | LOW_PRIORITY ] [IGNORE]
[INTO] table [ (column, ...) ]
VALUES ( values  [, values... ])

INSERT [LOW_PRIORITY] [IGNORE]
[INTO] table [ (column, ...) ]
SELECT ...

INSERT [LOW_PRIORITY] [IGNORE]
[INTO] table
SET column=value, column=value,...
```

Description

Inserts data into a table. The first form of this statement simply inserts the given values into the given columns. Columns in the table that are not given values are set to their default value or NULL. The second form takes the results of a SELECT query and inserts them into the table. The third form is simply an alternate version of the first form that more explicitly shows which columns correspond with which values. If the DELAYED modifier is present in the first form, all incoming SELECT statements will be given priority over the insert, which will wait until the other activity has finished before inserting the data. In a similar way, using the LOW_PRIORITY modifier with any form of INSERT will cause the insertion to be postponed until all other operations from the client have been finished.

When using a SELECT query with the INSERT statement, you cannot use the ORDER BY modifier with the SELECT statement. Also, you cannot insert into the same table you are selecting from.

Starting with MySQL 3.22.5 it is possible to insert more than one row into a table at a time. This is done by adding additional value lists to the statement separated by commas.

You must have INSERT privileges to use this statement.

Examples

```
# Insert a record into the 'people' table.
INSERT INTO people ( name, rank, serial_number )
VALUES ( 'Bob Smith', 'Captain', 12345 );
# Copy all records from 'data' that are older than a certain date into
# 'old_data'. This would usually be followed by deleting the old data from
# 'data'.
INSERT INTO old_data ( id, date, field )
SELECT ( id, date, field)
FROM data
WHERE date < 87459300;
# Insert 3 new records into the 'people' table.
INSERT INTO people (name, rank, serial_number )
VALUES ( 'Tim O\'Reilly', 'General', 1),
       ('Andy Oram', 'Major', 4342),
       ('Randy Yarger', 'Private', 9943);
```

KILL

Syntax

```
KILL thread_id
```

Description

Terminates the specified thread. The thread ID numbers can be found using the SHOW PROCESSES statement. Killing threads owned by users other than yourself require process privilege.

Example

```
# Terminate thread 3
KILL 3
```

LOAD

Syntax

```
LOAD DATA [LOCAL] INFILE file [REPLACE|IGNORE]
INTO TABLE table [delimiters] [(columns)]
```

Description

Reads a text file that is in a readable format and inserts the data into a database table. This method of inserting data is much quicker than using multiple `INSERT` statements. Although the statement may be sent from all clients just like any other SQL statement, the file referred to in the statement is assumed to be located on the server unless the `LOCAL` keyword is used.. If the filename does not have a fully qualified path, MySQL looks under the directory for the current database for the file.

With no delimiters specified, `LOAD DATA INFILE` will assume that the file is tab delimited with character fields, special characters escaped with the backslash (\), and lines terminated with a newline character.

In addition to the default behavior, you may specify your own delimiters using the following keywords:

`FIELDS TERMINATED BY '`*`c`*`'`

Specifies the character used to delimit the fields. Standard C language escape codes can be used to designate special characters. This value may contain more than one character. For example, `FIELDS TERMINATED BY ','` denotes a comma delimited file and `FIELDS TERMINATED BY '\t'` denotes tab delimited. The default value is tab delimited.

`FIELDS ENCLOSED BY '`*`c`*`'`

Specifies the character used to enclose character strings. For example, `FIELD ENCLOSED BY '"'` would mean that a line containing `"this, value", "this", "value"` would be taken to have three fields: `"this,value", "this", and "value"`. The default behavior is to assume that no quoting is used in the file.

`FIELDS ESCAPED BY '`*`c`*`'`

Specifies the character used to indicate that the next character is not special, even though it would usually be a special character. For example, with `FIELDS ESCAPED BY '^'` a line consisting of `First,Second^,Third,Fourth` would be parsed as three fields: `"First", "Second,Third"` and `"Fourth"`. The exceptions to this rule are the null characters. Assuming the `FIELDS ESCAPED BY` value is a backslash, `\0` indicates an ASCII `NUL` (character number 0) and `\N` indicates a MySQL `NULL` value. The default value is the backslash character. Note that MySQL itself considers the backslash character to be special. Therefore to indicate backslash in that statement you must backslash the backslash like this: `FIELDS ESCAPED BY '\\'`.

`LINES TERMINATED BY '`*`c`*`'`

Specifies the character that indicates the start of a new record. This value can contain more than one character. For example, with `LINES TERMINATED BY '.'`, a file consisting of `a,b,c.d,e,f.g,h,k.` would be parsed as three sepa-

rate records, each containing three fields. The default is the newline character. This means that by default, MySQL assumes that each line is a separate record.

The keyword FIELDS should only be used for the entire statement. For example:

```
LOAD DATA INFILE data.txt FIELDS TERMINATED BY ',' ESCAPED BY '\\'
```

By default, if a value read from the file is the same as an existing value in the table for a field that is part of a unique key, an error is given. If the REPLACE keyword is added to the statement, the value from the file will replace the one already in the table. Conversely, the IGNORE keyword will cause MySQL to ignore the new value and keep the old one.

The word NULL encountered in the data file is considered to indicate a null value unless the FIELDS ENCLOSED BY character encloses it.

Using the same character for more than one delimiter can confuse MySQL. For example, FIELDS TERMINATED BY ',' ENCLOSED BY ',' would produce unpredictable behavior.

If a list of columns is provided, the data is inserted into those particular fields in the table. If no columns are provided, the number of fields in the data must match the number of fields in the table, and they must be in the same order as the fields are defined in the table.

You must have SELECT and INSERT privileges on the table to use this statement.

Example

```
# Load in the data contained in 'mydata.txt' into the table 'mydata'. Assume
# that the file is tab delimited with no quotes surrounding the fields.
LOAD DATA INFILE 'mydata.txt' INTO TABLE mydata
# Load in the data contained in 'newdata.txt' Look for two comma delimited
# fields and insert their values into the fields 'field1' and 'field2' in
# the 'newtable' table.
LOAD DATA INFILE 'newdata.txt'
INTO TABLE newtable
FIELDS TERMINATED BY ','
( field1, field2 )
```

LOCK

Syntax

```
LOCK TABLES name
[AS alias] READ|WRITE [, name2 [AS alias] READ|WRITE, ...]
```

Description

Locks a table for the use of a specific thread. This command is generally used to emulate transactions. If a thread creates a `READ` lock all other threads may read from the table but only the controlling thread can write to the table. If a thread creates a `WRITE` lock, no other thread may read from or write to the table.

Using locked and unlocked tables at the same time can cause the process thread to freeze. You must lock all of the tables you will be accessing during the time of the lock. Tables you access only before or after the lock do not need to be locked. The newest versions of MySQL generate an error if you attempt to access an unlocked table while you have other tables locked.

Example

```
# Lock tables 'table1' and 'table3' to prevent updates, and block all access
# to 'table2'. Also create the alias 't3' for 'table3' in the current thread.
LOCK TABLES table1 READ, table2 WRITE, table3 AS t3 READ
```

OPTIMIZE

Syntax

```
OPTIMIZE TABLE name
```

Description

Recreates a table eliminating any wasted space. This is done by creating the optimized table as a separate, temporary table and then moving over to replace the current table. While the procedure is happening, all table operations continue as normal (all writes are diverted to the temporary table).

Example

```
OPTIMIZE TABLE mytable
```

REPLACE

Syntax

```
REPLACE INTO table [(column, ...)] VALUES (value, ...)

REPLACE INTO table [(column, ...)] SELECT select_clause
```

Description

Inserts data to a table, replacing any old data that conflicts. This statement is identical to INSERT except that if a value conflicts with an existing unique key, the new value replaces the old one. The first form of this statement simply inserts the given values into the given columns. Columns in the table that are not given values are set to their default value or NULL. The second form takes the results of a SELECT query and inserts them into the table.

Examples

```
# Insert a record into the 'people' table.
REPLACE INTO people ( name, rank, serial_number )
VALUES ( 'Bob Smith', 'Captain', 12345 )
# Copy all records from 'data' that are older than a certain date into
# 'old_data'. This would usually be followed by deleting the old data from
# 'data'.
REPLACE INTO old_data ( id, date, field )
SELECT ( id, date, field)
FROM data
WHERE date < 87459300
```

REVOKE

Syntax

```
REVOKE privilege [(column, ...)] [, privilege [(column, ...) ...]
ON table FROM user
```

Description

Removes a privilege from a user. The values of privilege, table, and user are the same as for the GRANT statement. You must have the GRANT privilege to be able to execute this statement.

SELECT

Syntax

```
SELECT [STRAIGHT_JOIN] [DISTINCT|ALL] value[, value2...]
[INTO OUTFILE 'filename' delimiters]
FROM table[, table2...] [clause]
```

Description

Retrieve data from a database. The SELECT statement is the primary method of reading data from database tables.

If you specify more than one table, MySQL will automatically join the tables so that you can compare values between the tables. In cases where MySQL does not perform the join in an efficient manner, you can specify `STRAIGHT_JOIN` to force MySQL to join the tables in the order you enter them in the query.

If the `DISTINCT` keyword is present, only one row of data will be output for every group of rows that is identical. The `ALL` keyword is the opposite of distinct and displays all returned data. The default behavior is `ALL`.

The returned values can be any one of the following:

Aliases
: Any complex column name or function can be simplified by creating an alias for it. The value can be referred to by its alias anywhere else in the `SELECT` statement (e.g., `SELECT DATE_FORMAT(date,"%W, %M %d %Y") as nice_date FROM calendar`).

Column names
: These can be specified as column, table.column or database.table.column. The longer forms are necessary only to disambiguate columns with the same name, but can be used at any time (e.g., `SELECT name FROM people; SELECT mydata.people.name FROM people`).

Functions
: MySQL supports a wide range of built-in functions (see later). In addition, user defined functions can be added at any time using the `CREATE FUNCTION` statement (e.g., `SELECT COS(angle) FROM triangle`).

By default, MySQL sends all output to the client that sent the query. It is possible however, to have the output redirected to a file. In this way you can dump the contents of a table (or selected parts of it) to a formatted file that can either be human readable, or formatted for easy parsing by another database system.

The `INTO OUTFILE 'filename'` modifier is the means in which output redirection is accomplished. With this the results of the `SELECT` query are put into *filename*. The format of the file is determined by the delimiters arguments, which are the same as the `LOAD DATA INFILE` statement with the following additions:

- The `OPTIONALLY` keyword may be added to the `FIELDS ENCLOSED BY` modifier. This will cause MySQL to thread enclosed data as strings and non-enclosed data as numeric.
- Removing all field delimiters (i.e., `FIELDS TERMINATED BY '' ENCLOSED BY ''`) will cause a fixed-width format to be used. Data will be exported according to the display size of each field. Many spreadsheets and desktop databases can import fixed-width format files.

The default behavior with no delimiters is to export tab delimited data using backslash (\) as the escape character and to write one record per line.

The list of tables to join may be specified in the following ways:

Table1, Table2, Table3, . . .

This is the simplest form. The tables are joined in the manner that MySQL deems most efficient. This method can also be written as `Table1 JOIN Table2 JOIN Table3, ...`. The `CROSS` keyword can also be used, but it has no effect (e.g., `Table1 CROSS JOIN Table2`) Only rows that match the conditions for both columns are included in the joined table. For example, `SELECT * FROM people, homes WHERE people.id=homes.owner` would create a joined table containing the rows in the people table that have id fields that match the owner field in the homes table.

Like values, table names can also be aliased (e.g., `SELECT t1.name, t2.address FROM long_table_name t1, longer_table_name t2`)

Table1 `STRAIGHT_JOIN` *Table2*

This is identical to the earlier method, except that the left table is always read before the right table. This should be used if MySQL performs inefficient sorts by joining the tables in the wrong order.

Table1 `LEFT [OUTER] JOIN` *Table2* `ON` *clause*

This checks the right table against the clause. For each row that does not match, a row of `NULL`s is used to join with the left table. Using the previous example `SELECT * FROM people, homes LEFT JOIN people, homes ON people.id=homes.owner`, the joined table would contain all of the rows that match in both tables, as well as any rows in the people table that do not have matching rows in the homes table, `NULL` values would be used for the homes fields in these rows. The `OUTER` keyword is optional and has no effect.

Table1 `LEFT [OUTER] JOIN` *Table2* `USING` *(column[, column2 . . .])*

This joins the specified columns only if they exist in both tables (e.g., `SELECT * FROM old LEFT OUTER JOIN new USING (id)`)

Table1 `NATURAL LEFT [OUTER] JOIN` *Table2*

This joins only the columns that exist in both tables. This would be the same as using the previous method and specifying all of the columns in both tables (e.g., `SELECT rich_people.salary, poor_people.salary FROM rich_people NATURAL LEFT JOIN poor_people`)

{ oj Table1 `LEFT OUTER JOIN` *Table2* `ON` *clause }*

This is identical to *Table1* `LEFT JOIN` *Table2* `ON` *clause* and is only included for ODBC compatibility. (The “oj” stands for “Outer Join”.)

If no clause is provided, SELECT returns all of the data in the selected tables.

The search clause can contain any of the following substatements:

`WHERE` *`statement`*
: The `WHERE` statement construct is the most common way of searching for data in SQL. This statement is usually a comparison of some type but can also include any of the functions listed below, except for the aggregate functions. Named values, such as column names and aliases, and literal numbers and strings can be used in the statement.

`GROUP BY` *`column`*`[, column2,...]`
: This gathers all of the rows together that contain data from a certain column. This allows aggregate functions to be performed upon the columns (e.g., `SELECT name,MAX(age) FROM people GROUP BY name`).

`HAVING` *`clause`*
: This is the same as a `WHERE` clause except that it is performed upon the data that has already been retrieved from the database. The `HAVING` statement is a good place to perform aggregate functions on relatively small sets of data that have been retrieved from large tables. This way, the function does not have to act upon the whole table, only the data that has already been selected (e.g., `SELECT name,MAX(age) FROM people GROUP BY name HAVING MAX(age)>80`).

`ORDER BY` *`column`* `[`*`ASC`*`|`*`DESC`*`][, `*`column2`* `[`*`ASC`*`|`*`DESC`*`],...]`
: Sorts the returned data using the given column(s). If `DESC` is present, the data is sorted in descending order, otherwise ascending order is used. Ascending order can also be explicitly stated with the `ASC` keyword (e.g., `SELECT name, age FROM people ORDER BY age DESC`).

`LIMIT [`*`start`*`,] `*`rows`*
: Returns Only the specified number of rows. If the start value is supplied, that many rows are skipped before the data is returned. The first row is number 0 (e.g., `SELECT url FROM links LIMIT 5,10` (returns URL's numbered 5 through 14).

`PROCEDURE` *`name`*
: In early versions of MySQL, this does not do anything. It was provided to make importing data from other SQL servers easier. Starting with MySQL 3.22, this substatement lets you specify a procedure that modifies the query result before returning it to the client.

SELECT supports the concept of functions. MySQL defines several built-in functions that can operate upon the data in the table, returning the computed value(s)

to the user. With some functions, the value returned depends on whether the user wants to receive a numerical or string value. This is regarded as the "context" of the function. When selecting values to be displayed to the user, only text context is used, but when selecting data to be inserted into a field, or to be used as the argument of another function, the context depends upon what the receiver is expecting. For instance, selecting data to be inserted into a numerical field will place the function into a numerical context. MySQL functions are detailed in full in Chapter 18.

Examples

```
# Find all names in the 'people' table where the 'state' field is 'MI'.
SELECT name FROM people WHERE state='MI'
# Display all of the data in the 'mytable' table.
SELECT * FROM mytable
```

SET

Syntax

```
SET OPTION SQL_OPTION=value
```

Description

Defines an option for the current session. Values set by this statement are not in effect anywhere but the current connection, and they disappear at the end of the connection. The following options are current supported:

`CHARACTER SET` *`charsetname or`* `DEFAULT`
: Changes the character set used by MySQL. Specifying `DEFAULT` will return to the original character set.

`LAST_INSERT_ID=`*`number`*
: Determines the value returned from the `LAST_INSERT_ID()` function.

`SQL_BIG_SELECTS=0` *`or`* `1`
: Determines the behavior when a large `SELECT` query is encountered. If set to 1, MySQL will abort the query with an error if the query would probably take too long to compute. MySQL decides that a query will take too long it will have to examine more rows than the value of the `max_join_size` server variable. The default value is 0, which allows all queries.

`SQL_BIG_TABLES=0` *`or`* `1`
: Determines the behavior of temporary tables (usually generated when dealing with large data sets). If this value is 1, temporary tables are stored on disk, which is slower than primary memory but can prevent errors on systems with low memory. The default value is 0, which stores temporary tables in RAM.

SQL_LOG_OFF=0 *or* 1

When set to 1, turns off standard logging for the current session. This does not stop logging to the ISAM log or the update log. You must have PROCESS LIST privileges to use this option. The default is 0, which enables regular logging.

SQL_SELECT_LIMIT=*number*

The maximum number of records returned by a SELECT query. A LIMIT modifier in a SELECT statement overrides this value. The default behavior is to return all records.

SQL_UPDATE_LOG=0 *or* 1

When set to 0, turns off update logging for the current session. This does not affect standard logging or ISAM logging. You must have PROCESS LIST privileges to use this option. The default is 1, which enables regular logging.

TIMESTAMP=*value* or DEFAULT

Determines the time used for the session. This time is logged to the update log and will be used if data is restored from the log. Specifying DEFAULT will return to the system time.

Example

```
# Turn off logging for the current connection.
SET OPTION SQL_LOG_OFF=1
```

SHOW

Syntax

```
SHOW COLUMNS FROM table [FROM database] [LIKE clause]
SHOW DATABASES [LIKE clause]
SHOW FIELDS FROM table [FROM database] [LIKE clause]
SHOW GRANTS
SHOW INDEX FROM table [FROM database]
SHOW KEYS FROM table [FROM database]
SHOW PROCESSLIST
SHOW STATUS
SHOW TABLE STATUS [FROM database [LIKE clause]]
SHOW TABLES [FROM database [LIKE expression]]
SHOW VARIABLES [LIKE clause]
```

Description

Displays various information about the MySQL system. This statement can be used to examine the status or structure of almost any part of MySQL.

Examples

```
# Show the available databases
SHOW DATABASES
# Display information on the indexes on table 'bigdata'
SHOW KEYS FROM bigdata
# Display information on the indexes on table 'bigdata'
# in the database 'mydata'
SHOW INDEX FROM bigdata FROM mydata
# Show the tables available from the database 'mydata' that begin with the
# letter 'z'
SHOW TABLES FROM mydata LIKE 'z%'
# Display information about the columns on the table 'skates'
SHOW COLUMNS FROM stakes
# Display information about the columns on the table 'people'
# that end with '_name'
SHOW FIELDS FROM people LIKE '%\_name'
# Show server status information.
SHOW STATUS
# Display server variables
SHOW VARIABLES
```

UNLOCK

Syntax

```
UNLOCK TABLES
```

Description

Unlocks all tables that were locked using the `LOCK` statement during the current connection.

Example

```
# Unlock all tables
UNLOCK TABLES
```

UPDATE

Syntax

```
UPDATE [LOW_PRIORITY] table
SET column=value, ...
[WHERE clause]
[LIMIT n]
```

Description

Alters data within a table. This statement is used to change actual data within a table without altering the table itself. You may use the name of a column as a value when setting a new value. For example, `UPDATE health SET miles_`

`ran=miles_ran+5` would add five to the current value of the miles_ran column. The statement returns the number of rows changed.

You must have `UPDATE` privileges to use this statement.

Example

```
# Change the name 'John Deo' to 'John Doe' everywhere in the people table.
UPDATE people SET name='John Doe' WHERE name='John Deo'
```

USE

Syntax

USE *database*

Description

Selects the default database. The database given in this statement is used as the default database for subsequent queries. Other databases may still be explicitly specified using the `database.table.column` notation.

Example

```
# Make db1 the default database.
USE db1
```