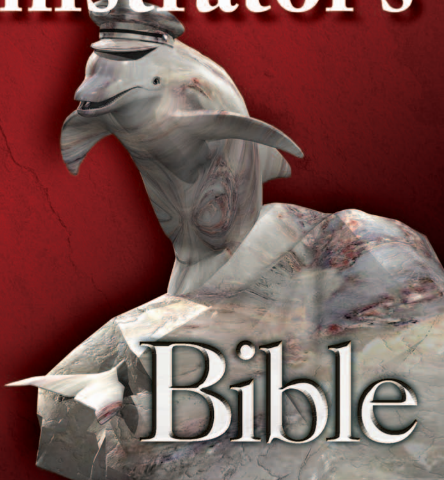
Companion Web Site

Sheeri K. Cabral and Keith Murphy

MySQL® Administrator's Bible



Learn best practices for administering MySQL

Transition from other databases to MySQL

Optimize MySQL queries and schemas

The book you need to succeed!

MySQL® Administrator's Bible

Chapter 4: How MySQL Extends and Deviates from SQL

ISBN: 978-0-470-41691-4

Sheeri K. Cabral
Keith Murphy

Copyright of Wiley Publishing, Inc.
Posted with Permission

Chapter 4

How MySQL Extends and Deviates from SQL

IN THIS CHAPTER

- Learning MySQL language Structure
- Understanding MySQL deviations
- Using MySQL extensions

MySQL was originally designed with three basic ideas in mind: to be fast, reliable, and easy to use. Like every database system, MySQL does not completely follow the SQL standard and has its own extensions to SQL. Some very basic SQL standards, however, did not exist for a very long time — the data dictionary was only added in MySQL 5.0, released in October 2005.

MySQL became popular because of its relative ease of use. For example, the SQL standard way of finding out which tables are in the `sakila` database is:

```
SELECT TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA='sakila';
```

The easiest way to find out this information in MySQL is:

```
SHOW TABLES FROM sakila;
```

Most of the differences from standard SQL and SQL extensions in MySQL come from the desire to make the database system easy to use. In the past several years, there has been more of a commitment to support the current SQL standard — as of the time of the writing of this book, that standard is the ANSI/ISO SQL:2003 standard. When we refer to "the SQL standard" or "standard SQL," we are referring to ANSI/ISO SQL:2003.

For information on the different SQL modes supported, see Chapter 5.

Using the Sample Database

In Chapter 3, you imported the sakila sample database. Throughout this chapter and the rest of the book, we will be using the sakila sample database to demonstrate database features. The sakila sample database was created by Mike Hillyer and contains data that might be found in a video rental store. For the purpose of our examples, we refer to a fictional video store whose data is stored in the sakila database and whose owner, Ziesel Skelley, uses a MySQL backend to store and query her company's information.

Learning MySQL Language Structure

Before getting into the extensions to SQL and deviations from SQL, there are some important rules to know about some of the language structure in MySQL, specifically relating to:

- Comments and portability
- Case-sensitivity
- Escape characters
- Naming limitations
- Quoting
- Time zones
- Character sets and collations

While some of these may be seen as SQL deviations or extensions, they are important enough to discuss before learning about the rest of MySQL's customized SQL syntax.

ON the WEBSITE **If you are new to SQL, there is an SQL Primer on the accompanying website for this book at www.wiley.com/go/mysqladminbible.**

Comments and portability

One of the cleverest MySQL extensions to the SQL standard is actually a way to manage portability. In some cases, portability of the schema and queries is desired, but being able to use MySQL-specific extensions when the environment is right is also desired. MySQL has an extension that specifies a MySQL version as part of a comment. The comment will be only parsed if the mysqld server is of an appropriate version; otherwise, the comment will be left as a comment and ignored by other databases and mysqld versions.

The -- is the standard SQL simple comment introducer. Everything on a line after this is considered a comment. The SQL standard bracketed comment introducer and terminator /* */ allow

partial line and multi-line commenting. Putting ! after the bracketed comment introducer indicates that this is MySQL specific code, and the mysqld server will parse it:

```
/*! SHOW DATABASES */;
```

The mysqld server will parse the SHOW DATABASES statement, but other database systems will not — that is, if the other database systems follow the SQL standard for comments. A five-digit number after the ! can be used to specify a minimum mysqld version. The first digit of the number is the major version, the next two digits are the minor version, and the last two digits are the revision number. For example, the output of mysqldump sakila starts with:

```
-- MySQL dump 10.13  Distrib 6.0.8-alpha
--
-- Host: localhost    Database: sakila
-- ------------------------------------------------------
-- Server version       6.0.8-alpha

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,
FOREIGN_KEY_CHECKS=0
*/;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_
ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;
```

This means that mysqld versions 4.1.1 and higher will apply the first statement, and mysqld versions 4.1.11 and higher will apply the last statement. A comment that will only be applied to mysqld versions 6.0.8 and higher begins with /*!60008.

While this syntax makes the code portable, it does not guarantee that the code will work as expected. For example, mysqldump exports tables in alphabetical order. The SET FOREIGN_KEY_CHECKS=0 statement is utilized so that a table with a foreign key constraint can be imported even if the referencing table has not yet been created. Attempting to import this on a different database server should result in an error, because a reference is being generated to a table that does not exist.

MySQL extends commenting by parsing anything after an unquoted, unescaped # as a comment, regardless of whitespace around the #:

```
mysql> SELECT#example of # as a comment
    -> 0;
```

```
+---+
| 0 |
+---+
| 0 |
+---+
1 row in set (0.00 sec)
```

See the "Escape characters" and "Naming limitations and quoting" sections in this chapter for more information on escaping and quoting characters.

Case-sensitivity

Traditionally, SQL reserved words are written in uppercase, such as `SELECT`, `FROM`, `NULL`, and `AS`. These words are case-insensitive, meaning that `SELECT`, `select`, and `SeLeCt` are all parsed by `mysqld` as the same reserved word. Throughout this book, we will format reserved words in uppercase, even though `mysqld` will properly parse reserved words regardless of case.

In general, `mysqld` is case-insensitive with respect to names of fields, indexes, stored routines and events. However, `mysqld` stores information on the file system and makes assumptions based on the files and directories found on the file system. For example, `mysqld` assumes that any directory in the data directory (`datadir`) is a database. This means that a directory with a path of `datadir/backup` will show up as a database named `backup` when `SHOW DATABASES` is run — whether or not `backup` was intended to be a database. All tables must have a `.frm` format file within the directory that represents the database they belong to; otherwise, `mysqld` does not recognize the table as existing.

MyISAM tables are defined by a `.frm` format file, a `.MYD` data file, and a `.MYI` index file (see Chapter 11 for more information on MyISAM tables). It is possible to move a MyISAM table to a different database by moving the associated files to a different directory. It is also possible to rename a MyISAM table by renaming the associated files. However, it is not recommended that you make these types of changes. To change the database of the `move_me` table from `test` to `test2`, issue the statement:

```
ALTER TABLE test.move_me RENAME test2.move_me;
```

To change the name of the `move_me` table to `keep_me`, issue:

```
ALTER TABLE move_me RENAME keep_me;
```

NOTE **It is not necessarily harmful to have a directory appear as a database (such as a `lost+found` directory). However, this may lead to errors with code that automatically traverse all databases, such as a custom backup script. It is important to be extremely careful when deleting and moving directories and files within the data directory and dropping databases.**

The way `mysqld` stores information on the file system has the consequence that some names are case-sensitive, if the file system is case-sensitive. Unix has a case-sensitive file system, and

Windows has a case-insensitive file system. Mac OS X uses a case-insensitive file system by default, although it supports other, case-sensitive file systems. Case-sensitivity for names is determined by whether or not the file system is case-sensitive for the following:

- Databases
- Tablespaces
- Tables
- Views
- Aliases
- Triggers
- Log file groups

In general, there should not be problems unless the databases are used on different file systems, for example when migrating from one file system to another. The static, global `lower_case_table_names` system variable can be set to change the default behavior of `mysqld`:

- If `lower_case_table_names=0` is set, table and database names are stored using the case specified in the `CREATE` statement. In queries, table names, table aliases, and database names are case-sensitive. This is the default value on Unix, where the file system is case-sensitive.
- If `lower_case_table_names=1` is set, table and database names are stored using lowercase. In queries, table names, table aliases, and database names are case-insensitive — they are actually converted to lowercase by `mysqld`. This is the default value on Windows, where the file system is case-insensitive.
- If `lower_case_table_names=2` is set, most table and database names are stored using the case specified in the `CREATE` statement. The exception is InnoDB table names, which are stored using lowercase. In queries, table names, table aliases, and database names are case-insensitive — they are converted to lowercase by `mysqld`, as they are when the value is set to 1. This is the default value on Mac OS X.

TIP **Come up with a naming convention that includes rules about case. For example, perhaps databases and tables will be all lowercase, with underscores to show different words (such as in `table_name`) and fields will use *camelCase* (such as in `fieldName`). In this way, the issue of case is resolved, no matter what platform is being used.**

Escape characters

The escape character is the backslash (`\`). An *escape sequence* is a backslash followed by one character. Chapter 3 discussed some commands that looked like escape sequences such as `\P` and `\u` — these are actually translated into other commands by the `mysql` command line client. However, `mysqld` has actual escape sequences, which are independent of the `mysql` commands.

These escape sequences are used only within strings (the mysql ones are not within strings) and are parsed unless the NO_BACKSLASH_ESCAPES SQL mode is set. See Chapter 5 for more information on SQL modes.

The escape sequences for strings in mysqld are:

- \\ to print the \ character.
- \' to print the ' character, even if the string is quoted with '.
- \" to print the " character, even if the string is quoted with ".
- _ prints the _ character. This can be used to search for the actual value when using LIKE. If _ is not escaped, it is used as the wildcard character for one character.
- \% prints the % character. This can be used to search for the actual value % when using LIKE. If % is not escaped, it is used as the wildcard character for one or more characters.
- \b prints a backspace character, which actually means that it deletes the previous character. For example, SELECT "Hi!\b" returns Hi, not Hi! because of the backspace.
- \r and \n print a carriage return and a new line, respectively.
- \t prints a tab separator, as actually hitting the Tab key may try to use the auto-complete function of the mysql client. For more information on the auto-complete function in mysql, see Chapter 3.
- \0 and \Z print ASCII 0 (NUL) and ASCII 26 (Ctrl-Z), respectively. In Windows, Ctrl-Z is a special character marking an end of a file, and on Mac OS X and Unix, Ctrl-Z is the special key sequence that will suspend the current foreground process.

These escape sequences are case-sensitive. In a string, a backslash followed by any other character will just print the character. The following queries exemplify this behavior:

```
mysql> SELECT 'Hi!';
+-----+
| Hi! |
+-----+
| Hi! |
+-----+
1 row in set (0.00 sec)

mysql> SELECT 'Hi!\b';
+------+
| Hi  |
+------+
| Hi  |
+------+
1 row in set (0.00 sec)
```

```
mysql> SELECT 'Hi!\B';
+------+
| Hi!B |
+------+
| Hi!B |
+------+
1 row in set (0.00 sec)
```

Naming limitations and quoting

Identifiers are names of: databases, tables, views, fields, indexes, tablespaces, stored routines, triggers, events, servers, log file groups, and aliases (specified with the AS keyword). Identifiers are all limited to 64 characters except aliases, which are limited to 255 characters. Note that characters may be one or more bytes; see Chapter 5 for a discussion of the difference between characters and bytes. Identifiers are stored using the utf8 character set.

WARNING **The utf8 character set before MySQL 6.0 used up to 3 bytes to store each character. In MySQL 6.0 and up, that character set was renamed to utf8mb3, and the character set named utf8 is a true implementation of Unicode, using up to 4 bytes to store each character. When upgrading from MySQL 5.1 to MySQL 6.0, fields using utf8 follow the renaming of the character set and are shown to use utf8mb3. There is no conversion that takes place during an upgrade from MySQL 5.1 to MySQL 6.0. Therefore, to use a true implementation of Unicode, MySQL must be upgraded to version 6.0 or higher, and an explicit conversion must be done of all fields using utf8mb3. For more information about this type of upgrade, see the MySQL manual at** http://dev.mysql.com/doc/refman/6.0/en/charset-unicode-upgrading.html.

Identifiers can be almost anything. However, identifiers may not end with one or more spaces:

```
mysql> CREATE TABLE `space ` (id INT);
ERROR 1103 (42000): Incorrect table name 'space '
mysql> CREATE TABLE space (`id ` INT);
ERROR 1166 (42000): Incorrect column name 'id '
```

Identifier names can be reserved words or numbers, and include punctuation. However, to be parsed correctly, such identifiers need to be quoted. When sql_mode includes ANSI_QUOTES (see "SQL Modes" in Chapter 5), the double quotation mark character (") is used to quote identifiers such as database and table names, and strings are quoted by the single quotation mark character (').When sql_mode does not include ANSI_QUOTES, as is the default, the backtick character (`) is used to quote identifiers such as database and table names, and strings are quoted by the either the double quotation mark character (") or the single quotation mark character (').

The escape string is the backslash (\) — see the "Escape characters" section earlier in this chapter for more information. This is used in conjunction with strings to escape special

characters. For example, % and _ are wildcard characters as specified in the SQL standard. To find a string that contains the actual character % using the LIKE operator, escape it with \:

```
mysql> USE sakila;
Database changed
mysql> SELECT first_name FROM staff WHERE first_name LIKE 'M%';
+------------+
| first_name |
+------------+
| Mike       |
+------------+
1 row in set (0.00 sec)

mysql> SELECT first_name FROM staff WHERE first_name LIKE 'M\%';
```

The empty set (0.00 sec) in queries, strings needs to be quoted to distinguish it from an identifier. The following example shows that, when the string Mike is not quoted, mysqld parses it as the name of a field. However, when the string is quoted, mysqld parses it as a string:

```
mysql> SELECT last_name FROM staff WHERE first_name=Mike;
ERROR 1054 (42S22): Unknown column 'Mike' in 'where clause'
mysql> SELECT last_name FROM staff WHERE first_name='Mike';
+-----------+
| last_name |
+-----------+
| Hillyer   |
+-----------+
1 row in set (0.00 sec)
```

On the other hand, numbers will be parsed as numbers. In order to parse a number as an identifier, it must be quoted. The following example shows that when the number 1 is not quoted, mysqld parses it as a number. However, when the number is quoted, mysqld parses it as a field name:

```
mysql> SELECT first_name, last_name FROM staff WHERE active=1;
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Mike       | Hillyer   |
| Jon        | Stephens  |
+------------+-----------+
2 rows in set (0.00 sec)

mysql> SELECT first_name, last_name FROM staff WHERE active=`1`;
ERROR 1054 (42S22): Unknown column '1' in 'where clause'
```

Note that a number can be any of the following:

- An unsigned integer such as 12345
- A signed integer such as +12345 or -12345

- A signed or unsigned decimal such as 12.345, +12.345 or -12.345
- A signed or unsigned number in scientific notation format, such as 123.4e+5, 123.4e-5, or -123.4e5
- A number in binary format, such as 0b100 (the number 4)
- A number in hexadecimal format, such as 0x100 or 0x1F4 (the numbers 256 and 500, respectively)
- If used in a numeric context, a string beginning with a number will be parsed as a number equal to the first numeric part of the string. For example, '10q4' is parsed as 10 when used in a numeric context:

```
mysql> SELECT 1+'10q4';
+----------+
| 1+'10q4' |
+----------+
|       11 |
+----------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------+
| Level   | Code | Message                                 |
+---------+------+-----------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: '10q4' |
+---------+------+-----------------------------------------+
1 row in set (0.00 sec)
```

- If such a truncation occurs when storing a value in a table, by default mysqld issues a similar warning and stores the truncated value. For mysqld to issue an error and refuse to store such values, a strict SQL mode must be used. See Chapter 5 for more information on SQL modes. Errors only apply to storing values in tables; the preceding example will always issue a warning and calculate the expression with the truncated value, even if a strict SQL mode is used.
- An expression that results in a number, such as 1+1 or ABS(-1)

Like numbers, the reserved words TRUE, FALSE, and NULL do not need to be quoted. As with other reserved words, TRUE, FALSE, and NULL are case-insensitive.

Dot notation

MySQL has a special *dot notation* that can be used to specify a database when referring to a table. Simply place a dot character (.) between the database and table name. The following example shows that the TABLES table does not exist in the current working database (sakila) but prefixing the table name with the string "INFORMATION_SCHEMA." specifies the database to which the TABLES table belongs:

```
mysql> SELECT TABLE_NAME FROM TABLES LIMIT 1;
ERROR 1146 (42S02): Table 'sakila.tables' doesn't exist
```

```
mysql> SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES LIMIT 1;
+----------------+
| TABLE_NAME     |
+----------------+
| CHARACTER_SETS |
+----------------+
1 row in set (0.01 sec)
```

MySQL allows identifiers such as tables and fields to have any name, including reserved words, such as `FROM`, and function names, such as `COUNT` and `DATE`. Furthermore, table names can use any character, including punctuation such as `.` and `;`. Reserved words and punctuation other than _ must be enclosed by the quotation mark that specifies an identifier; this is usually the backtick (`) character (see the "Naming limitations and quoting" section earlier in this chapter for more information). Function names do not need to be enclosed by quotation marks. The following example shows that `mysqld` returns an error if reserved words (`FROM`) and special punctuation (`.`) are not enclosed by the backtick character, but function names (`DATE`) do not need to be enclosed similarly to work:

```
mysql> CREATE TABLE name.test(from DATE);
ERROR 1049 (42000): Unknown database 'name'
mysql> CREATE TABLE `name.test`(from DATE);
ERROR 1064 (42000): You have an error in your SQL syntax; check
the manual that corresponds to your MySQL server version for the
right syntax to use near 'from DATE)' at line 1
mysql> CREATE TABLE `name.test`(date DATE);
Query OK, 0 rows affected (0.37 sec)

mysql> SELECT COUNT(*) FROM dot.test;
ERROR 1146 (42S02): Table 'dot.test' doesn't exist
mysql> SELECT COUNT(*) FROM test.dot.test;
ERROR 1064 (42000): You have an error in your SQL syntax; check
the manual that corresponds to your MySQL server version for the
right syntax to use near '.test' at line 1
mysql> SELECT COUNT(*) FROM test.`dot.test`;
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
1 row in set (0.01 sec)
```

Identifier names can be a numbers, such as a table named `12345`, but they must be quoted as an identifier just as reserved words and punctuation are. This dot notation also extends to fields:

```
mysql> SELECT test.`dot.test`.foo FROM test.`dot.test` limit 1;
Empty set (0.00 sec)

mysql> SELECT `dot.test`.foo FROM test.`dot.test` limit 1;
Empty set (0.00 sec)
```

Note that the table name `dot.test` had to be quoted with backticks. Using dot notation to specify fields is not done very often, but it can be useful when querying fields with similar names.

TIP **Avoid problematic naming of your databases, tables, and fields. Do not use a number, reserved word, function name, or punctuation other than _ in your database, table, and field names. If you are getting an error because - is being parsed as the subtraction function, use _ instead. Because `mysqld` requires quoting for numbers, reserved words, and punctuation, those are easily avoided. It is more difficult to realize when function names such as `COUNT` and `DATE` are used.**

Time zones

When `mysqld` starts, it determines the time zone of the operating system and sets the `system_time_zone` system variable accordingly. By default, `mysqld` sets the value of `time_zone` to `SYSTEM`, which means that it operates using the time zone in `system_time_zone`. Fields with a `TIMESTAMP` value are converted to UTC and stored. When retrieved, `TIMESTAMP` fields are converted to the value of `time_zone`. Because `TIMESTAMP` fields are stored as UTC values and conversion is transparent, there are no problems if the underlying operating system time zone changes.

Because of the many issues surrounding time zones, it is best to keep the `time_zone` variable set to a value of `SYSTEM`. Even when `mysqld` uses the operating system time zones, problems can arise in the following scenarios:

- Unlike `TIMESTAMP`, `DATE`, `TIME`, and `DATETIME` fields are not converted and stored as UTC. Furthermore, there is no time zone information in these fields. If a `DATETIME` value is stored as `'2009-01-01 00:00:00'`, there is no way to specify what the time zone associated with that time is. For other differences between `TIMESTAMP` and other time-based data types, see Chapter 5.
- Web servers and database servers may have different time zones, and thus have different times. An order coming from an application on a web server whose time is in PST (UTC-8) and stored on a database server whose time is in EST (UTC-5) has a problem: At what time was the order made? Was it made on December 31, 2008 at 9 PM PST or January 1, 2009 at midnight EST? While both of those represent the same time, recall that `DATE`, `TIME`, and `DATETIME` fields do not store time zone information. For taxes, tariffs, legal purposes, and financial reports, does that order belong in 2008 or 2009?
- If the application code uses the time local to the web server as the order time, orders from web servers configured with different time zones will be stored in the database as if they occurred at different times. For example, if two different web servers are configured so that one uses PST and the other uses EST, orders coming in at the exact same time — January 1, 2009 at 5 AM UTC — and stored in a `DATETIME` field will appear to have been made three hours apart, because the time zone information will be lost.

- To avoid this issue, do not have application code determine the time. Instead, use the `CURRENT_TIMESTAMP()` function or the `NOW()` alias to insert the current time into a field. These values are replication-safe (do not use the `SYSDATE()` function; it is not safe for replication).

NOTE **If you feel up to the challenge of trying to manage time zones within MySQL and can brave adding another level of complexity to time management, the MySQL manual has information on how to specify time zones in MySQL at:**

```
http://dev.mysql.com/doc/refman/6.0/en/time-zone-support.html
```

In practice, managing time zones in MySQL adds so much headache and hassle that is rarely worth it. In our experience, it is much more worthwhile to set operating systems to the same time zone (preferably UTC) and configure time synchronization software (such as `ntpd`) so that all servers have the same times and time zones.

Character sets and collations

MySQL supports many different *character sets* and *collations*. A character set, or *charset*, is the set of available characters that can be used — similar to an alphabet. Different languages have different alphabets, and the most often used character sets contain the letters of many alphabets (for example, the default latin1 character set includes all of the characters in Latin languages, including accented characters and characters using the cedilla. A collation specifies the lexical sort order; in English the lexical sort order begins with `a`, `b`, `c`, `d`; in Spanish the lexical sort order begins with `a`, `b`, `c`, `ch`, `d`; in Greek the lexical sort order begins with α, β, χ, δ. A collation can also specify if a sort order is case-sensitive or not; a *binary collation* is a collation that is case-sensitive. In MySQL, binary collations usually end in `_bin`, such as `ascii_bin`. In a binary collation, the sort order is determined by the numeric representation of the character. This has the result that sort order is case-sensitive, because the same letter in a different case has a different numeric representation.

To illustrate the difference among case-insensitive sort order, case-sensitive sort order, and binary sort order, create a table that specifies three fields, each with its own sort order, and insert some values:

```
mysql> use test;
Database changed
mysql> CREATE TABLE sort_test (
    -> ci CHAR(1) CHARACTER SET latin1 COLLATE latin1_general_ci,
    -> cs CHAR(1) CHARACTER SET latin1 COLLATE latin1_general_cs,
    -> cbin CHAR(1) CHARACTER SET latin1 COLLATE latin1_bin);
Query OK, 0 rows affected (0.39 sec)
```

```
mysql> INSERT INTO sort_test VALUES
    -> ('A','A','A'), ('a','a','a'), ('b','b','b'), ('B','B','B');
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

In case-insensitive search, first the letters a and A are displayed, and then the letters b and B are displayed:

```
mysql> SELECT ci FROM sort_test ORDER BY ci;
+------+
| ci   |
+------+
| A    |
| a    |
| b    |
| B    |
+------+
4 rows in set (0.00 sec)
```

Note that the order with regard to case does not matter — in this case, the order of the input determines the order of the output. A case-sensitive ordering, on the other hand, sorts the letters a and A before b and B, but also sorts with respect to capital letters — within a letter, the capital letter is sorted first:

```
mysql> SELECT cs FROM sort_test ORDER BY cs;
+------+
| cs   |
+------+
| A    |
| a    |
| B    |
| b    |
+------+
4 rows in set (0.00 sec)
```

In a binary collation, the order depends on the order of the numeric representation. The numeric order of the letters in the preceding example is:

```
mysql> SELECT ASCII('a'),ASCII('A'),ASCII('b'),ASCII('B');
+------------+------------+------------+------------+
| ASCII('a') | ASCII('A') | ASCII('b') | ASCII('B') |
+------------+------------+------------+------------+
|         97 |         65 |         98 |         66 |
+------------+------------+------------+------------+
1 row in set (0.00 sec)
```

Therefore, a binary collation should sort these letters as A B a b, because numerically the order is 65 66 97 98. And indeed, this is the case:

```
mysql> SELECT cbin FROM sort_test ORDER BY cbin;
+------+
| cbin |
+------+
| A    |
| B    |
| a    |
| b    |
+------+
4 rows in set (0.00 sec)
```

Many other database systems support only one character set, utf8. The default character set in MySQL is latin1, and the default collation is latin1_swedish_ci. Each character set has at least one collation, and each collation is associated with exactly one character set. Collation names are of the form *charset_description_suffix*, where *charset* is the character set (such as latin1), *description* is a brief description (such as swedish or general), and *suffix* is either ci (case insensitive), cs (case sensitive), or bin (binary).

NOTE **The exception to the collation naming rule is the binary collation, which is associated with the binary character set and is a binary collation. If it conformed to the naming algorithm, its name would be binary_binary_bin.**

Each character set has a default collation and may have other collations associated with it as well. The default character set for MySQL, latin1, has a default collation of latin1_swedish_ci and seven additional collations:

```
mysql> SELECT COLLATION_NAME, IS_DEFAULT
    -> FROM INFORMATION_SCHEMA.COLLATIONS
    -> WHERE CHARACTER_SET_NAME='latin1';
+-------------------+------------+
| COLLATION_NAME    | IS_DEFAULT |
+-------------------+------------+
| latin1_german1_ci |            |
| latin1_swedish_ci | Yes        |
| latin1_danish_ci  |            |
| latin1_german2_ci |            |
| latin1_bin        |            |
| latin1_general_ci |            |
| latin1_general_cs |            |
| latin1_spanish_ci |            |
+-------------------+------------+
8 rows in set (0.00 sec)
```

For more information on the COLLATIONS system view in the INFORMATION_SCHEMA database, see Chapter 21.

All collations disregard trailing whitespace in sort ordering. Some data types automatically strip trailing whitespace; see Chapter 5 for more details.

The default character set and collation in MySQL can be set at many different levels. Changing the character set without changing the collation will result in the character set's default collation to be used. Changing the collation without changing the character set will result in the character set being changed to the character set associated with the specified collation.

Standard SQL defines the `CHARACTER SET` and `COLLATE` clauses in `CREATE TABLE` and `ALTER TABLE` statements only when adding fields. Specifying a character set and collation on a table level is not supported by standard SQL. When specifying character sets on any level in MySQL, `CHARSET` is an alias for `CHARACTER SET`.

The different levels to which character set and collation can be set are:

- **Server** — The system variables `character_set_server` and `collation_server` specify the default character set and collation for a database when a `CREATE DATABASE` statement does not have any `CHARACTER SET` or `COLLATE` clauses.
- **Database** — The system variables `character_set_database` and `collation_database` specify the default character set and collation for the current database. These are set with the `CHARACTER SET` and `COLLATE` clauses of the `CREATE DATABASE` and `ALTER DATABASE` statements. The database character set and collation are used by `LOAD DATA INFILE` and specify the default character set and collation for a table when a `CREATE TABLE` statement does not have any `CHARACTER SET` or `COLLATE` clauses.
- The session variables will change the current database only, and the global variables will change all databases. When checking the default character set for the current database, make sure that you are in the correct database and are looking at the correct variable scope (`GLOBAL` or `SESSION`).

WARNING **The `SHOW VARIABLES` and `SET` commands default to using session-level variables. To avoid confusion, always use `SHOW SESSION VARIABLES` or `SHOW GLOBAL VARIABLES` and `SET GLOBAL`, `SET @@global`, `SET SESSION`, or `SET @@session`.**

- Database options such as the default character set and collation are stored in plain text in the `db.opt` file in each database.
- **Table** — A `CREATE TABLE` or `ALTER TABLE tblname ADD COLUMN` statement can use a `CHARACTER SET` or `COLLATE` clause. This will set the default character set and collation for a field added with no character set or collation specified.
- **Field** — The earlier example with the `sort_test` table showed how to specify `CHARACTER SET` and `COLLATE` clauses to fields that are the `CHAR`, `VARCHAR`, `TINYTEXT`, `TEXT`, `MEDIUMTEXT` or `LONGTEXT` data type. For more information on data types, see Chapter 5. The syntax for the `CHARACTER SET` and `COLLATE` clauses is the same for `CREATE TABLE` and `ALTER TABLE` statements. Each field can have its own character set and collation, as shown in the example with the `sort_test` table.

NOTE **Most of the time, individual fields are not given different character sets and collations. Make sure that your default character sets and collations are set appropriately.**

- **String** — The system variables `character_set_connection` and `collation_connection` specify the default character set and collation for the strings sent via the current connection. A string such as the one in `SELECT "hi"` will be returned with the character set and collation specified by the `character_set_connection` and `collation_connection` system variables.
- Standard SQL allows the character set of a string to be specified with an *introducer*, which is simply the underscore character (_) followed by the character set name. The introducer appears before the string:

```
mysql> SELECT 'hi', CHARSET('hi');
+----+---------------+
| hi | CHARSET('hi') |
+----+---------------+
| hi | latin1        |
+----+---------------+
1 row in set (0.00 sec)

mysql> SELECT _ascii 'hi', CHARSET(_ascii 'hi');
+----+---------------------+
| hi | CHARSET(_ascii 'hi') |
+----+---------------------+
| hi | ascii               |
+----+---------------------+
1 row in set (0.00 sec)
```

- The introducer tells `mysqld` to parse the string using that character set. Similarly, the `COLLATE` clause, also standard SQL, tells `mysqld` to parse the string using the specified collation. Unlike the introducer, the `COLLATE` clause comes after the string:

```
mysql> SELECT COLLATION(_ascii 'hi'),
    -> COLLATION (_ascii 'hi' COLLATE ascii_bin)\G
*************************** 1. row ***************************
                   COLLATION(_ascii 'hi'): ascii_general_ci
COLLATION (_ascii 'hi' COLLATE ascii_bin): ascii_bin
1 row in set (0.00 sec)
```

- The introducer and the `COLLATE` clause are different from the `CAST` and `CONVERT` functions. The `CAST` and `CONVERT` functions take a string of one character set and collation and change it to another character set and collation. The introducer and `COLLATE` clause inform `mysqld` of the character set and collation of the string.
- The default collation of a result can be overridden by specifying an introducer and `COLLATE` clause for a string. For example, to override the `latin1_bin` collation on the `cbin` field of the `sort_test` table use:

```
mysql> SELECT cbin FROM sort_test
    -> ORDER BY cbin COLLATE latin1_general_ci;
+------+
| cbin |
+------+
| A    |
| a    |
| b    |
| B    |
+------+
4 rows in set (0.00 sec)
```

- This yields the same result as when the `ci` field was sorted using its default collation.
- The `character_set_client` variable specifies the character set used by the client application. Statements are converted from `character_set_client` to `character_set_connection`, unless an introducer and `COLLATE` clause are applied to a string. Often (and by default), the `character_set_client` and `character_set_connection` are set to the same value, so this conversion does not need to happen. However, in the event that a client uses an ASCII character set, connects to a database, and wants all statements to be converted to UTF-8, the `character_set_client` would be set to `ascii` and the `character_set_connection` would be set to `utf8`. The conversions will be done automatically by `mysqld`. These two variables ensure that clients using different character sets can send the same queries to `mysqld` and achieve the same results.
- The `character_set_results` variable is similar to the `character_set_client` variable. However, the `character_set_results` variable sets the character set that results should be returned as. If a retrieved field has a character set of `latin1` and the `character_set_results` value is `utf8`, `mysqld` will convert the retrieved field to `utf8` before sending the result back to the client.

NOTE **To summarize the confusing connection/client/results relationship, consider this: A client sends a statement in `character_set_client`, which is converted to `character_set_connection` and `collation_connection` by `mysqld`. After query execution, results are converted by `mysqld` to `character_set_results`. Almost all of the time, the `character_set_client` and `character_set_results` variables will be set with the same value — the character set of the client.**

- **File system** — The `character_set_filesystem` system variable specifies the character set of the file system. When a filename is specified (such as in a `LOAD DATA INFILE` statement), `mysqld` converts the filename from the character set specified by `character_set_client` to the one specified by `character_set_filesystem`. The default value is `binary`, which means that no conversion is done and the filename is used as specified.

- **System —** The character_set_system variable is always set to utf8, as that is the character set mysqld uses to store information such as identifiers.

The SET CHARACTER SET charset_name statement sets the value of the session variables character_set_client and character_set_connection to charset_name for the duration of the session, or until the system variables are changed again.

The SET NAMES charset_name statement sets the value of the session variables character_set_client, character_set_connection, and character_set_results to charset_name for the duration of the session, or until the system variables are changed again. SET NAMES charset_name COLLATE collation_name additionally sets the session variable collation_connection to collation_name.

The charset charset_name command is a mysql client command, which behaves as a SET NAMES command, except that values will not be lost if a reconnect (such as with \r) occurs.

To have the mysql client behave as if it connects and immediately runs a SET NAMES charset_name statement, use the --default-character-set=charset_name option to mysql.

WARNING **Converting data from one character set to another can produce unexpected results. Consult the MySQL manual for more information about how to convert fields from one character set to another.**

Functions that manipulate strings will return a string in the same character set and collation as the input string. Functions do not change the character set or collation of a string unless explicitly told to do so (with CAST, CONVERT, BINARY, COLLATE, or the use of an introducer). Note that other conversions may occur, for instance, if mysqld changes the character set due to values of character_set_client, character_set_connection and character_set_results. Also, note that the REPLACE function always does a case-insensitive match, regardless of the collation of the strings involved.

MySQL uses the standard SQL CONVERT(text_expr USING charset_name) syntax. MySQL extends SQL with the CAST(text_expr AS text_data_type CHARACTER SET charset_name) syntax. For example:

```
mysql> SELECT CAST('hi' AS CHAR(2) CHARACTER SET ascii);
+-------------------------------------------+
| CAST('hi' AS CHAR(2) CHARACTER SET ascii) |
+-------------------------------------------+
| hi                                        |
+-------------------------------------------+
1 row in set (0.00 sec)
```

CAST cannot specify a collation. However, because CAST returns a string expression, a COLLATE clause can follow a CAST function to indicate what collation should be associated with the string. If the previous example wanted to specify that the return from the CAST function should

be considered the ascii_bin collation, the following SELECT statement would have been issued:

```
mysql> SELECT
    -> CAST('hi' AS CHAR(2) CHARACTER SET ascii) COLLATE ascii_bin;
```

For information on how to examine the available character sets and collations, see Chapter 21.

Understanding MySQL Deviations

MySQL has worked on supporting the ODBC SQL standard and the ANSI SQL standard. However, as with all other database systems, some of the SQL commands do not function as the standard indicates. For example, there are many privilege system differences in how the GRANT and REVOKE commands work; some deviations are listed in the "Privileges and Permissions" section. For more complete information on how the privilege system works in MySQL, see Chapter 14. How MySQL handles transactions and isolation levels is discussed in Chapter 9.

While this section explains deviations from theoretical SQL standards and some of the more outstanding differences from expected behavior, it does not explain all specific deviations from other database management systems, such as Oracle, Microsoft SQL Server, Sybase, or DB2. There are many details that SQL standards do not explain, and many of these details are implementation details. For instance, materialized views are an implementation detail of how views are handled by a database management system. The SQL standard covers syntax to create and drop views, but it does not specify that views should be materialized. However, if you are experienced in a database management system that uses materialized views, you may assume that a view in MySQL is materialized. This assumption can lead to poor query performance. As much as possible, we have tried to explain how MySQL works, particularly with respect to implementation details that are radically different from other database management systems. For example, Chapter 8 discusses views and explains that MySQL does not use materialized views, showing examples of how MySQL parses and optimizes view queries.

MySQL deviates from how most database administrators expect it to behave in the following ways:

- **Storage engines** — Each table is an instantiation of a storage engine. Different tables can have different storage engines. Different storage engines function differently with regard to performance, ACID (atomicity, consistency, isolation, durability) compliance (see Chapter 9 for more information on ACID compliance), supported features, and more. Information about how different storage engines work appears throughout the book; however, Chapter 11 focuses on the major differences among the storage engines.
- **Errors** — MySQL makes attempts to make sense of what should throw an error. By default, mysqld automatically allows inserting invalid data, automatically truncates data that is too large for a data type, implicitly converts data and more. The sql_mode server variable can be set to change most of this type of behavior. See Chapter 5 for more details on SQL modes.

- **String comparison —** By default, strings are compared in the order determined by the collation (see the "Character sets and collations" section earlier in this chapter for more information). However, strings are compared in a case-insensitive manner, unless a string is cast using the `BINARY()` function or a string is stored in a field that specifies the `BINARY` attribute. See Chapter 5 for more details on specifying the `BINARY` attribute in a field that stores strings.
- In addition, `LIKE` can be used to compare numbers:

```
mysql> SELECT 0 LIKE 0;
+----------+
| 0 LIKE 0 |
+----------+
|        1 |
+----------+
1 row in set (0.00 sec)
```

MySQL deviates from the SQL standard in the following ways:

- **Data types —** See Chapter 5 for how MySQL deviates from the SQL standard data types.
- **Index types —** See Chapter 6 for how MySQL deviates from the SQL standard index types.
- MySQL has no refined concept of `GLOBAL` or `LOCAL` tables. A `TEMPORARY` table can be considered local, as it is an in-memory table that is only available to the session that defines it. All other tables can be considered global, as they are available to other sessions as soon as a successful `CREATE TABLE` statement completes.
- MySQL does not support the `ON COMMIT`, `REF IS`, `UNDER` and `AS SUBQUERY` clauses to `CREATE TABLE`. The `LIKE` clause to `CREATE TABLE` only takes a table name as an argument; no options (such as `INCLUDING DEFAULTS` and `EXCLUDING DEFAULTS`) are allowed.
- `SCHEMA` is an alias to `DATABASE` in the `CREATE SCHEMA`, `ALTER SCHEMA` and `DROP SCHEMA` statements. MySQL does not support the `AUTHORIZATION` clause.
- Stored code such as the stored procedures, stored functions, triggers, and events can only use SQL statements. MySQL has no equivalent to a procedural programming language such as PL/SQL. Stored code in MySQL does not contain many of the options in the SQL standard and contains a few extensions. See Chapter 7 for more information about stored procedures, stored functions, triggers, and events.
- User-defined system views (metadata) are allowed, though they can only be written only in C. See the "Custom Metadata section" in Chapter 21 for more information.
- User-defined functions are allowed, though they can only be written only in C.

ON the WEBSITE **More information on how to create and use a user-defined function can be found on the accompanying website for this book at `www.wiley.com/go/mysqladminbible`.**

- Comparison operators can be used in the result expressions of the `SELECT` fields. If used, these will return `TRUE` (1), `FALSE` (0), or `NULL` (`NULL`). For example:

```
mysql> use sakila;Database changed
mysql> SELECT staff_id, staff_id>1, '|',
    -> first_name, first_name LIKE 'M%'
    -> FROM staff;
+----------+------------+---+------------+----------------------+
| staff_id | staff_id>1 | | | first_name | first_name LIKE 'M%' |
+----------+------------+---+------------+----------------------+
|        1 |          0 | | | Mike       |                    1 |
|        2 |          1 | | | Jon        |                    0 |
+----------+------------+---+------------+----------------------+
2 rows in set (0.00 sec)
```

- MySQL does not support the concept of catalogs. The INFORMATION_SCHEMA database has many system views with fields relating to catalogs where the value is NULL. In addition, the SET CATALOG statement is not supported.
- **Foreign key constraints** — MySQL accepts foreign key constraints in table definitions, but only tables using transactional storage engines (such as InnoDB and Falcon) actually implement foreign key checking. All other storage engine types disregard foreign key constraint definitions without producing an error. See Chapter 6 for more details on foreign key constraints in MySQL.
- In MySQL, foreign key constraints are always checked, unless the FOREIGN_KEY_CHECKS session variable is set to OFF or 0. There is no way to disable foreign key constraints on an individual basis. Thus, MySQL does not support the REFERENCES ARE CHECKED and REFERENCES ARE NOT CHECKED clauses.
- The MATCH clause in a foreign key constraint is disregarded in all storage engines, even InnoDB and Falcon, which implement foreign keys.
- Prepared statements in MySQL have a local scope; that is, a prepared statement can only be used by the session that creates it. MySQL does not have as big a performance boost when using prepared statements, because each session has to compile each prepared statement the first time it is used in the session. In addition, the PREPARE syntax in MySQL is very basic — it only allows the name of the prepared statement and the statement itself to be specified (using a string or a user-defined variable containing a string). PREPARE in MySQL does not support the ATTRIBUTES keyword.
- MySQL does not support querying data samples using the TABLESAMPLE clause.
- CAST() and CONVERT() — According to the SQL standard, the CAST() and CONVERT() functions should be able to cast to any data type. However, in MySQL, CAST()and CONVERT() cannot be used to cast a number to the REAL or BIGINT data type.
- In standard SQL, the TRIM() function can only be used to remove a single leading character and a single trailing character. In MySQL, the TRIM() function can be used to remove multiple leading and trailing characters.
- **String concatenation** — The || string concatenation function is not supported. Use CONCAT() instead. In MySQL, || is an alias for OR.

- MySQL does not support assertions. The CREATE ASSERTION and DROP ASSERTION statements are not supported.
- MySQL does not support the SQL standard way of defining character sets and collations. The CREATE CHARACTER SET, DROP CHARACTER SET, CREATE COLLATION, and DROP COLLATION statements are not supported. For details on how to add a character set to MySQL, see the manual page at:

  http://dev.mysql.com/doc/refman/6.0/en/adding-character-set.html
- For details on how to add a collation to MySQL, see the manual page at:

  http://dev.mysql.com/doc/refman/6.0/en/adding-collation.html
- GROUP BY in MySQL does not support the CUBE or GROUPING SETS options.
- MySQL does not support the following functions: binary set functions (including CORR, COVAR_POP, COVAR_SAMP, etc.), COLLECT, FUSION, and INTERSECTION.
- MySQL does not support windowing functions such as RANK, DENSE_RANK, PERCENT_RANK, CUME_DIST.
- MySQL does not support static cursors. In MySQL, all cursors are dynamic (prepared at runtime). However, cursors are stored (cached) in temporary tables, so they are not fully dynamic. The performance of cursors in MySQL is usually worse than the performance of cursors in other database management systems. See http://forge.mysql.com/worklog/task.php?id=3433 for more details on the tasks that need to be completed in order for cursors to be fully dynamic.
- MySQL does not support domains or domain constraints. The CREATE DOMAIN, ALTER DOMAIN, and DROP DOMAIN statements are not supported, and DOMAIN permissions cannot be granted using GRANT.
- MySQL does not support sequences. The CREATE SEQUENCE, ALTER SEQUENCE, and DROP SEQUENCE statements are not supported, and SEQUENCE permissions cannot be granted using GRANT. In addition, field definitions in MySQL do not support the GENERATED, ALWAYS, BY DEFAULT, and AS IDENTITY keywords. The LIKE clause of a CREATE TABLE statement in MySQL does not support the INCLUDING IDENTITY and EXCLUDING IDENTITY options.
- MySQL does not support user-defined types nor transform functions for user-defined types. The CREATE CAST, DROP CAST, CREATE ORDERING FOR, DROP ORDERING FOR, CREATE TYPE, ALTER TYPE, DROP TYPE, CREATE TRANSFORM, ALTER TRANSFORM, and DROP TRANSFORM statements are not supported, and TYPE permissions cannot be granted using GRANT.
- MySQL does not support transliterations. The CREATE TRANSLATION and DROP TRANSLATION statements are not supported, and TRANSLATION permissions cannot be granted using GRANT.
- MySQL does not support any embedded declarations. DECLARE is supported for cursors but not for embedded SQL, embedded authorization declarations, and temporary table

declarations. Temporary tables can be created by specifying CREATE TEMPORARY TABLE instead of CREATE TABLE and dropped with DROP TEMPORARY TABLE. MySQL extends DECLARE to be able to specify variables, conditions, and handlers. See Chapter 7 for more details on the DECLARE extensions.

- Updatable cursors and the WHERE CURRENT OF clauses in UPDATE and DELETE statements are not supported in MySQL.
- MySQL does not support recursive queries or the SEARCH DEPTH FIRST BY, SEARCH BREADTH FIRST BY, and CYCLE clauses.
- In the SQL standard, DESCRIBE is used to obtain information about prepared statement input and output parameters. In MySQL, DESCRIBE is an alias for SHOW COLUMNS. Table 4-1 shows the DESCRIBE syntax and how it is translated into SHOW COLUMNS statements.

TABLE 4-1

Translating DESCRIBE into SHOW COLUMNS

DESCRIBE Statement	Corresponding SHOW COLUMS Statement
DESCRIBE tblname;	SHOW COLUMNS FROM tblname;
DESCRIBE tblname fldname;	SHOW COLUMNS FROM tblname WHERE Field='fldname';
DESCRIBE tblname 'fldname';	SHOW COLUMNS FROM fldname WHERE Field LIKE 'fldname';

- The third syntax can be used with the % and _ wildcard characters. DESC can be used in place of DESCRIBE, if desired. For more information on SHOW COLUMNS, see Chapter 21.
- MySQL does not support descriptor areas. The ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, and SET DESCRIPTOR statements are not supported.
- MySQL does not support connection management with the CONNECT TO, SET CONNECTION, and DISCONNECT statements. In addition, session management is not supported — the SET ROLE and SET TIME ZONE statements are not supported by MySQL. The SQL standard SET SESSION statement is not supported by MySQL. However, MySQL has a conflicting syntax — the SET statement takes an optional keyword of GLOBAL or SESSION when setting a system variable. Therefore, even though SQL standard SET SESSION commands such as SET SESSION AUTHORIZATION, and SET SESSION CHARACTERISTICS are not valid, there are valid nonstandard SET SESSION commands, such as:

```
SET SESSION character_set_client=latin1;
```

- MySQL does not support the SET SCHEMA statement. The default database can be set by specifying a default database as a client option when connecting, and it can be changed in the mysql client program with the \u or use command.

- MySQL does not support dynamically prepared statements using `EXECUTE IMMEDIATE`. Regular prepared statements are supported, as is the `EXECUTE` statement without the `IMMEDIATE` qualifier.
- MySQL does not support diagnostics management with the `GET DIAGNOSTICS` statement. The `SHOW ERRORS` and `SHOW WARNINGS` statements can be used to see errors and warnings from the previous statement, and the error log can be monitored for errors (and warnings if the `log_warnings` system variable is set). More information on `SHOW ERRORS` and `SHOW WARNINGS` can be seen later in this chapter in The "SHOW extension" section.

Privileges and permissions

MySQL uses the `GRANT` and `REVOKE` syntax as specified in the SQL standard, with some changes already mentioned (such as lack of domains and thus a lack of `DOMAIN` privileges) in addition to the following deviations:

- There is no `WITH ADMIN OPTION`; instead the `SUPER` privilege exists, and the `WITH GRANT OPTION` can be specified to allow a user to `GRANT` any subset of privileges that user has to another user.
- MySQL does not support the `GRANTED BY` or `WITH HIERARCHY OPTION` clauses.
- MySQL has a limited concept of users; a user is unique with respect to its `username@host` value. However, because of wildcards, `localhost`, and multiple hostname support, it is possible that a user connecting from a particular host may not receive the expected permissions. Fields cannot be associated with a user, role or path.
- MySQL does not support the concept of roles. The `DROP ROLE` statement is not supported.
- One or more users can be renamed using the `RENAME USER` statement:

```
RENAME USER user1@host1 TO user2@host2;
RENAME USER user1@host1 TO user2@host2, userA@hostA TO userB@hostB;
```

- A user can be created without having any privileges explicitly granted via a `CREATE USER user@host [ IDENTIFIED BY 'password_string' ]` statement. The `USAGE ON *.*` privilege is implicitly granted by this statement.
- `DROP USER user@host` will revoke all privileges, including `USAGE`, from `user@host`.
- There are no `CHARACTER SET` or `COLLATION` privileges.

For more information about privileges and permissions, see Chapter 14.

Transaction management

Transaction management is partially supported in MySQL. Transactions are only supported when using tables defined with transactional storage engines, such as InnoDB and Falcon.

For more information on storage engines, see Chapter 11; for more information on transactions in MySQL, see Chapter 9.

MySQL supports the `START TRANSACTION` command to note the beginning of a transaction. However, `START TRANSACTION` in MySQL does not allow any optional arguments to specify transaction modes. Table 4-2 shows the SQL standard transaction modes and how those transaction modes can be set in MySQL.

TABLE 4-2

Setting Transaction Modes in MySQL

SQL Standard	MySQL Equivalent
`ISOLATION LEVEL iso_level`	See Chapter 9 for how to set isolation levels.
`READ ONLY`	Change permissions for the user. See Chapter 14 for more information on granting privileges. Set the server to `read_only`. See the "Promoting a new master" section in Chapter 22 for more information on the `read_only` server variable.
`READ WRITE`	Change permissions for the user; see Chapter 14 for more information on granting privileges.
`DIAGNOSTICS SIZE`	N/A (MySQL does not support this feature with an alternative syntax)

`SET TRANSACTION` and `SET LOCAL TRANSACTION` commands are not supported by MySQL.

Check constraints

MySQL does not support check constraints, other than those implemented by specifying data types, foreign key constraints, and unique key constraints (for more information about key constraints, see Chapter 6). The `SET CONSTRAINTS` statement is not supported.

Check constraints defined with the `CONSTRAINT...CHECK` clause in the `CREATE TABLE` or `ALTER TABLE` statements are allowed but ignored, no matter what storage engine is used. The following example defines a check constraint where the `id` field must only have a value of 0, and shows how the check constraint is ignored:

```
mysql> CREATE TABLE check_test (id INT PRIMARY KEY) ENGINE=InnoDB;
Query OK, 0 rows affected (0.37 sec)

mysql> ALTER TABLE check_test
    -> ADD CONSTRAINT is_ignored CHECK (id=0);
Query OK, 0 rows affected (0.39 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> SHOW CREATE TABLE check_test\G
*************************** 1. row ***************************
       Table: check_test
Create Table: CREATE TABLE `check_test` (
  `id` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)

mysql> INSERT INTO check_test (id) VALUES (0),(1);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT id FROM check_test;
+----+
| id |
+----+
|  0 |
|  1 |
+----+
2 rows in set (0.00 sec)
```

Upsert statements

MySQL does not support *upsert* statements with the standard SQL `MERGE` statement. An upsert statement either inserts a new record or, if a certain condition is met, updates existing records. MySQL supports limited upsert statements with the `ON DUPLICATE KEY UPDATE` clause to an `INSERT` statement. The SQL standard `MERGE` statement supports any condition, but the `ON DUPLICATE KEY UPDATE` clause in MySQL only supports the condition where a unique or primary key already exists.

To test this, first find a suitable key constraint on the `store` table in the `sakila` database, and some data to work with:

```
mysql> SELECT INDEX_NAME, SEQ_IN_INDEX, COLUMN_NAME
    -> FROM INFORMATION_SCHEMA.STATISTICS
    -> WHERE NON_UNIQUE=0 AND TABLE_SCHEMA='sakila'
    -> AND TABLE_NAME='store';
+-------------------+--------------+------------------+
| INDEX_NAME        | SEQ_IN_INDEX | COLUMN_NAME      |
+-------------------+--------------+------------------+
| PRIMARY           |            1 | store_id         |
| idx_unique_manager |           1 | manager_staff_id |
+-------------------+--------------+------------------+
2 rows in set (0.00 sec)

mysql> use sakila
Database changed
mysql> SELECT store_id, manager_staff_id, address_id, last_update
```

```
    -> FROM store;
+----------+------------------+------------+---------------------+
| store_id | manager_staff_id | address_id | last_update         |
+----------+------------------+------------+---------------------+
|        1 |                1 |          1 | 2006-02-15 04:57:12 |
|        2 |                2 |          2 | 2006-02-15 04:57:12 |
+----------+------------------+------------+---------------------+
2 rows in set (0.00 sec)
```

Ziesel (the store owner of our fictional video store rental company that utilizes the `sakila` database) wants to make sure that all of her stores are in the database. The data to upsert (insert or update records) is shown in Table 4-3.

TABLE 4-3

Data to Upsert to Store

Store_id	Manager_staff_id	Address_id
1	1	1
2	2	3

As Ziesel feared, the data in the `store` table is not correct. Specifically, the `address_id` for store 2 is incorrect. Table 4-3 corresponds with the following upsert statement, which should update the `address id` for the record with a `store_id` of 2:

```
mysql> INSERT INTO store (store_id, manager_staff_id, address_id)
    -> VALUES (1,1,1),(2,2,3)
    -> ON DUPLICATE KEY UPDATE address_id=VALUES(address_id);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 1  Warnings: 0

mysql> SELECT store_id, manager_staff_id, address_id, last_update
 FROM store;
+----------+------------------+------------+---------------------+
| store_id | manager_staff_id | address_id | last_update         |
+----------+------------------+------------+---------------------+
|        1 |                1 |          1 | 2006-02-15 04:57:12 |
|        2 |                2 |          3 | 2009-01-25 04:35:18 |
+----------+------------------+------------+---------------------+
2 rows in set (0.00 sec)
```

As desired, the record with a `store_id` of 2 was updated.

Similar upsert behavior can be accomplished with the `REPLACE` statement. A `REPLACE` statement, like the `ON DUPLICATE KEY UPDATE` statement, will insert a record if there is no existing

record with a duplicate PRIMARY or UNIQUE KEY constraint. However, if there is a duplicate key constraint, REPLACE will DELETE the existing record and INSERT a new one. This has many consequences — two actions are performed (DELETE and INSERT) instead of one (UPDATE, as with ON DUPLICATE KEY UPDATE). In an ON DUPLICATE KEY UPDATE statement, any INSERT or any UPDATE trigger may fire, depending on whether data was inserted or updated. In a REPLACE statement, either the INSERT trigger(s) or the INSERT and DELETE triggers will fire — either the record is inserted, in which case the INSERT trigger(s) will fire, or the record is replaced via DELETE and INSERT statements, in which case those triggers will fire.

WARNING REPLACE **can be useful, but keep in mind that performance may suffer because there are two actions being performed.**

The REPLACE statement has almost the same syntax as the INSERT statement, including the LOW_PRIORITY and DELAYED extensions (see the "DML extensions" section later in this chapter). However, the IGNORE and ON DUPLICATE KEY UPDATE extensions are not part of REPLACE syntax. This is acceptable, as the desired behavior when a duplicate key is found is either:

- Delete the existing record and insert a new record (as with REPLACE)
- Update the existing record (as with ON DUPLICATE KEY UPDATE)
- Do nothing (as with IGNORE)

Thus, the REPLACE, ON DUPLICATE KEY UPDATE, and IGNORE clauses are mutually exclusive.

Using MySQL Extensions

Many of the extensions to MySQL have been developed to make MySQL easier to use. Many of the MySQL SHOW statements, for example, are much easier to use for a beginner than having to query the INFORMATION_SCHEMA database. Some of the extensions to SQL are actually commands in the client, such as use (\u) and source (\.) and have been discussed in Chapter 3. Many of the extensions are in mysqld. The following extensions are explained throughout the rest of this chapter:

- Aliases
- ALTER TABLE extensions
- CREATE extensions
- DML extensions (INSERT, UPDATE, DELETE)
- DROP extensions
- The LIMIT extension
- SELECT extensions

- Server maintenance extensions
- The SET extension
- The SHOW extension
- Table definition extensions
- Table maintenance extensions
- Transactional statement extensions

ON the WEBSITE **MySQL has added numerous new functions and extensions to existing functions. Appendix B contains a reference of all of the functions and their syntaxes. On the accompanying website for this book at `www.wiley.com/go/mysqladminbible`, you will find examples and uses for the functions with nonstandard behaviors, and the more frequently used nonstandard functions.**

Aliases

The following MySQL extensions are aliases to standard SQL statements:

- `BEGIN` and `BEGIN WORK` are aliases for `START TRANSACTION`.
- `DROP PREPARE stmt_prep` is a synonym for `DEALLOCATE PREPARE stmt_prep`.
- `EXPLAIN tbl_name` is an alias for `SHOW COLUMNS FROM tbl_name`.
- `num1 % num2` is the same as `MOD(num1,num2)`.

ALTER TABLE extensions

The `ALTER TABLE` statement has a number of extensions in MySQL that add features to allow an `ALTER TABLE` statement to do almost everything a `CREATE TABLE` statement can do. Many `ALTER TABLE` statements are *offline* statements — to change the table, the statements copy the table, blocking access to the table, as if the table itself were offline. The largest performance enhancement is that many `ALTER TABLE` statements are *online* statements — `ALTER TABLE` statements that do not copy the table. Online statements are done in the background.

The following statements are online statements:

- `ADD INDEX` and `DROP INDEX` for variable-width indexed fields.
- Renaming a field using `CHANGE COLUMN` and specifying the same data type.
- Using `CHANGE COLUMN` or `MODIFY COLUMN` to modify the default value for a field.
- Adding items to the end of an `ENUM` or `SET` value data type (with `CHANGE COLUMN` or `MODIFY COLUMN`). See Chapter 5 for more information about data types, including the `ENUM` and `SET` data types.

Unfortunately, that still leaves many operations as offline operations. However, a MySQL extension to `ALTER TABLE` that can help that is the ability to specify more than one operation at a

time on a table, using a comma-separated list of ALTER TABLE options. For example, Ziesel uses the following query on the film table in the sakila database to perform three offline operations at the same time: adding a new field for the film's country of origin with a default country of the United States (country_id 103), creating an index on that field, and creating a foreign key constraint on that field to the country_id field of the country table.

```
mysql> use sakila;
Database changed
mysql> ALTER TABLE film
    -> ADD COLUMN origin_country SMALLINT(5) UNSIGNED
    ->   NOT NULL DEFAULT 103,
    -> ADD INDEX idx_fk_origin_country (origin_country),
    -> ADD CONSTRAINT fk_film_country FOREIGN KEY (origin_country)
    -> REFERENCES country(country_id);
Query OK, 1000 rows affected (1.12 sec)
Records: 1000  Duplicates: 0  Warnings: 0
```

Other than check constraints, which MySQL does not handle (see the "Understanding MySQL Deviations" section earlier in this chapter), the SQL standard defines the following actions an ALTER TABLE can perform:

- ADD/ALTER/DROP COLUMN
- ADD/DROP PRIMARY/UNIQUE/FOREIGN KEY

MySQL has added many extensions that add functionality to ALTER TABLE and provide methods to control the performance of ALTER TABLE:

- ADD FULLTEXT INDEX — Add a fulltext index. See Chapter 6 for more details on fulltext indexes.
- ADD INDEX — Add an index. See Chapter 6 for more details on indexes.
- ADD SPATIAL INDEX — Add a spatial index.

ON the WEBSITE **For more information about spatial indexes, see the companion website for this book at www.wiley.com/go/mysqladminbible.**

- CHANGE COLUMN old_fld_name new_fld_name new_fld_definition — Change the field name and definition. Note that there is no way to change the field name without specifying the field definition as well. In addition, the field definition can end with either FIRST or AFTER other_fld_name to specify the position the field should be put in.
- CONVERT TO CHARACTER SET charset_name
- CONVERT TO CHARACTER SET charset_name COLLATION collation_name
- DISABLE KEYS — Disables any indexes so that they are not updated when records are inserted, deleted, or updated. Speeds up large data imports in conjunction with ENABLE KEYS.

- ENABLE KEYS — Enables automatic index updating and rebuilds all indexes on the table. Speeds up large data imports in conjunction with DISABLE KEYS.
- IGNORE — If an ALTER TABLE statement results in a duplicate key error, the table copy is stopped and the table is reverted to its original schema. All of the changes in the ALTER TABLE are lost, even if the change did not cause the duplicate key error. When you specify IGNORE between ALTER and TABLE, duplicate records that would cause such errors are deleted from the table.

  To see this behavior, Ziesel copies her customer table:

```
mysql> use sakila;
Database changed
mysql> CREATE TABLE customer_test LIKE customer;
Query OK, 0 rows affected (0.04 sec)

mysql> INSERT INTO customer_test SELECT * FROM customer;
Query OK, 599 rows affected (0.17 sec)
Records: 599  Duplicates: 0  Warnings: 0
```

  Now that she has a table with all 599 customers that she can test without destroying her production data, Ziesel purposefully causes a duplicate key error, so that she can later compare ALTER TABLE to ALTER IGNORE TABLE:

```
mysql> SELECT COUNT(*), active
    -> FROM customer_test
    -> GROUP BY active;
+----------+--------+
| COUNT(*) | active |
+----------+--------+
|       15 |      0 |
|      584 |      1 |
+----------+--------+
2 rows in set (0.02 sec)

mysql> ALTER TABLE customer_test ADD UNIQUE KEY(active);
ERROR 1062 (23000): Duplicate entry '1' for key 'active'
```

  Now that she has caused a duplicate key error, she compares the behavior of using the IGNORE keyword:

```
mysql> ALTER IGNORE TABLE customer_test ADD UNIQUE KEY(active);
Query OK, 599 rows affected (0.40 sec)
Records: 599  Duplicates: 597  Warnings: 0

mysql> SELECT COUNT(*), active
    -> FROM customer_test
    -> GROUP BY active;
```

```
+----------+--------+
| COUNT(*) | active |
+----------+--------+
|        1 |      0 |
|        1 |      1 |
+----------+--------+
2 rows in set (0.00 sec)

mysql> SELECT COUNT(*) from customer_test;
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
1 row in set (0.00 sec)
```

There were 597 duplicate keys that were deleted because of the `ALTER IGNORE`. Only two records are left in the table — one record with an `active` value of 0, and the other with an `active` value of 1. Take care not to lose important data when using `ALTER IGNORE TABLE`.

- `MODIFY COLUMN fld_name new_fld_definition` — Note that there is no way to change a part of the field definition without specifying the whole field definition. For example, to change an `INT NOT NULL` to an `UNSIGNED INT NOT NULL`, the entire field definition `UNSIGNED INT NOT NULL` must be used. In addition, the field definition can end with either `FIRST` or `AFTER other_fld_name` to specify the position the field should be put in.
- `ORDER BY fld_list` — Performs a one-time sort of the data records, sorting each row in order of the comma-separated field list (just as if it was the result of a `SELECT` query with the same `ORDER BY` clause).
- `RENAME new_tblname` or `RENAME TO new_tblname` will change the name of a table and associated objects such as triggers and foreign key constraints.

Other table-level extensions are listed in the "Table definition extensions" section later in this chapter. Table extensions are valid for both `CREATE TABLE` and `ALTER TABLE` statements. For example, `ENGINE=MyISAM` is valid for both `CREATE TABLE` and `ALTER TABLE`:

```
CREATE TABLE foo (id int) ENGINE=MyISAM
ALTER TABLE foo ENGINE=MyISAM
```

CREATE extensions

Many MySQL `CREATE` statements contain an `IF NOT EXISTS` extension. This specifies that a warning, not an error, should be issued if `mysqld` cannot complete the `CREATE` statement because of an existing identifier conflict. For example:

```
mysql> CREATE DATABASE IF NOT EXISTS test;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+-------+------+------------------------------------------------+
| Level | Code | Message                                        |
+-------+------+------------------------------------------------+
| Note  | 1007 | Can't create database 'test'; database exists |
+-------+------+------------------------------------------------+
1 row in set (0.00 sec)
```

- Creating an index in a `CREATE TABLE` statement is a MySQL extension to standard SQL. In addition, creating a named index, specifying an index storage method (such as `USING HASH`) and creating an index that uses a column prefix are also nonstandard SQL — whether the index is created with `CREATE INDEX` or `ALTER TABLE ADD INDEX`. See Chapter 6 for more details on all of the standard and nonstandard features of indexes in MySQL.
- `CREATE VIEW` can be specified as `CREATE OR REPLACE VIEW view_name` to create a view if a view with `view_name` does not exist, or delete the existing view and replace it with the new view being defined if it does exist.
- Other table-level extensions are listed in the "Table definition extensions" section later in this chapter. Table extensions are valid for both `CREATE TABLE` and `ALTER TABLE` statements. For example, the `ENGINE=MyISAM` is valid for both of these:

```
CREATE TABLE foo (id int) ENGINE=MyISAM
ALTER TABLE foo ENGINE=MyISAM
```

DML extensions

MySQL extends DML (Data Manipulation Language — `INSERT`, `REPLACE`, `UPDATE`, and `DELETE` statements) with the following:

- `IGNORE` — Any errors caused by executing the specified DML are issued as warnings. This will cause the statement to continue instead of stopping at the first error. All errors appear as warnings and can be seen by issuing `SHOW WARNINGS` after the DML finishes.
- `LOW_PRIORITY` — Does not receive a write lock and execute the specified DML (`INSERT/REPLACE/UPDATE/DELETE`) until all read locks have been granted and there are no locks waiting in the read lock queue. (The default behavior is for all write locks to be granted before any read locks). The `LOW_PRIORITY` option is specified just after the first word of the statement — for example, `INSERT LOW_PRIORITY INTO tblname`.

  The `low-priority-updates` option to `mysqld` changes the default behavior so that all DML acts as if it were specified with `LOW_PRIORITY`. In other words, the

low-priority-updates option changes the default behavior to grant all read locks before granting a write lock.

If the low-priority-updates option is specified, the INSERT statement can take a HIGH_PRIORITY option to prioritize the write lock for specific INSERT statements. The HIGH_PRIORITY option is specified in the same position the LOW_PRIORITY option is. However, the HIGH_PRIORITY option is only valid with the INSERT statement — the LOW_PRIORITY statement is valid with all DML. Both LOW_PRIORITY and HIGH_PRIORITY only affect storage engines with table-level locks as their most granular lock.

See the "Table-level locks" section in Chapter 9 for more information on read and write lock queues.

- LIMIT — UPDATE and DELETE statements can change or delete a subset of matching rows. See "The LIMIT extension" section earlier in this chapter for details.
- ORDER BY — UPDATE and DELETE statements can specify a particular order. This is usually used with the LIMIT clause to change or delete only some rows — for example, ORDER BY and LIMIT can be used together in a SELECT statement to retrieve the oldest five records in a table. In the same way, ORDER BY and LIMIT can be used with UPDATE or DELETE to change or remove the oldest five records in a table.
- Upsert — MySQL has extended the INSERT statement to include upsert (insert/update) functionality. See the Upsert statements subsection (under the "Understanding MySQL deviations" section) earlier in this chapter for more information about upsert statements in MySQL, including the ON DUPLICATE KEY option to INSERT and the new REPLACE statement.
- DELETE QUICK — The QUICK option to DELETE may speed up some deletes by not merging index leaves when it changes the index to reflect that records have been removed. This can lead to more fragmentation in the index.
- TRUNCATE — Issue TRUNCATE tbl_name (or TRUNCATE TABLE tbl_name) to very quickly remove all the rows from a table. This does not actually issue any DELETE statements, so no DELETE triggers are invoked. Most storage engines drop and re-create the table; in addition to being faster than a DELETE statement, this will reset the AUTO_INCREMENT value to 0.

  InnoDB will drop and re-create the table unless there are foreign key constraints, in which case it will act exactly as DELETE FROM tbl_name, with no filter specified in a WHERE clause so all rows are deleted. If foreign keys are present, rows are deleted one at a time and foreign key ON DELETE clauses are processed as usual.

  Aside from the speed, another reason to use TRUNCATE instead of DELETE is if a DELETE cannot be used, for example when a table has a corrupt index or the data itself is corrupt. In addition, a DELETE statement requires the DELETE privilege, and a TRUNCATE statement requires the DROP privilege. Therefore, TRUNCATE can be used to remove all rows from a table if a user has the DROP privilege but not the DELETE privilege.

- `INSERT` readability — The `INSERT` statement has an alternate syntax for better readability when inserting many fields. This alternate syntax uses one or more `SET fld=value` clauses, like the standard syntax for `UPDATE`. The following two queries illustrate the difference between the SQL standard for `INSERT` statements (first query) and the alternative `INSERT` syntax allowed by MySQL (second query):

```
INSERT INTO address (address, address2, district, city_id,
postal_code, phone) VALUES
('44 Massachusetts Avenue', 'Apt. 102', 'Bergen County', 5,
'07742', '867-5309');

INSERT INTO address SET address='44 Massachusetts Avenue',
address2='Apt. 102', district='Bergen County', city_id=5,
postal_code='07742', phone='867-5309';
```

  Both queries are valid in MySQL and would insert the exact same row into the `address` table. Although it is longer, the second syntax makes it easier to correspond field names and the values being inserted. This also makes it very difficult to specify a different number of field names and values, such as in the following query (there is no value for the `phone` field):

```
INSERT INTO address (address, address2, district, city_id,
postal_code, phone) VALUES
('44 Massachusetts Avenue','Apt. 102', 'Bergen County', 5,
'07742');
ERROR 1136 (21S01): Column count doesn't match value count at row 1
```

- `DELETE` using more than one table — Alternate syntaxes for `DELETE` allow rows from multiple tables to be used in the deletion criteria, or allow rows from multiple tables to be deleted, or both. `ORDER BY` and `LIMIT` cannot be used when more than one table is specified, but the `LOW_PRIORITY`, `QUICK` and `IGNORE` options can be used.

  The syntaxes that allow `DELETE` to reference and/or delete from more than one table are:

```
DELETE tbl_list FROM tbl_expr [ WHERE condition ]
DELETE FROM tbl_list USING tbl_expr [ WHERE condition ]
```

  In both syntaxes, `tbl_list` is a comma-separated list of tables whose rows should be deleted based on the `tbl_expr` and the optional `WHERE` clause. The expression `tbl_expr` can be any expression that returns a table, including any type of `JOIN` clause and subqueries. Any tables that are in `tbl_expr` that are not in `tbl_list` will not have rows deleted.

- `INSERT DELAYED` — The `DELAYED` option to `INSERT` specifies that the data should be queued for a later batch insertion. When an `INSERT DELAYED` is issued, `mysqld` puts the information into a queue and returns successfully. The session can continue without waiting for the `INSERT` to finish. Many `INSERT DELAYED` statements are batched together and written at the same time, which is faster than many individual writes when there is a

lot of activity on the table. INSERT DELAYED will wait until there is no activity on the table and then insert a batch of records.

If there is not a lot of activity on a table, INSERT DELAYED will not perform better than individual INSERT statements. If there is not a lot of activity on a table when an INSERT DELAYED is issued, mysqld still puts the INSERT DELAYED information into a queue and returns successfully. However, the queue can immediately insert the batch in the queue. If the table has little activity, mysqld will be doing batch inserts where the batch size is 1 record. Regular INSERT statements would be faster in this case, because INSERT DELAYED has the additional overhead of enqueuing and dequeuing the information and the extra thread per table used to insert the batch. The MySQL manual has a detailed account of what takes place in an INSERT DELAYED statement at http://dev.mysql.com/doc/refman/6.0/en/insert-delayed.html.

INSERT DELAYED is not appropriate for data that needs to be stored in the database immediately. The batch queue is stored in memory, and in the event of a crash or a schema change from a higher priority ALTER TABLE statement, the information in the batch queue will be lost and *not* inserted. In addition, LAST_INSERT_ID() will not function as expected, because it reflects the most recent value actually inserted.

INSERT DELAYED can only be used on tables using the MyISAM, ARCHIVE, BLACKHOLE, and MEMORY storage engines and cannot be used on views or partitioned tables. The DELAYED option is ignored if an upsert is specified with ON DUPLICATE KEY, and when the SQL standard INSERT INTO...SELECT syntax is used.

- LOAD DATA INFILE — The LOAD DATA INFILE command is used to load data from a text file created by the SELECT INTO OUTFILE command. See the section on SELECT extensions for more information about SELECT INTO OUTFILE.

  To show an example of LOAD DATA INFILE first export the rental table from the sakila database, using SELECT ... INTO OUTFILE. By default, this puts the file in the directory of the database, but a location for the file can be specified optionally.

```
mysql> SELECT * FROM rental INTO OUTFILE 'rental.sql';
Query OK, 16044 rows affected (0.05 sec)
```

  There is no table definition included in the SELECT ... INTO OUTFILE so you should always ensure that you have a copy of the table definition for restoration of the file:

```
shell> mysqldump --no-data sakila rental > /tmp/rental-schema.sql
```

  To create a new database sakila2 and load the rental table definition into it:

```
shell> mysqladmin create sakila2
shell> mysql sakila2 < /tmp/rental-schema.sql
```

  Then, load the data into the sakila2.rental table:

```
mysql> use sakila2;
Database changed
```

```
mysql> LOAD DATA INFILE '/tmp/rental.sql' INTO TABLE rental;
Query OK, 16044 rows affected (1.24 sec)
Records: 16044  Deleted: 0  Skipped: 0  Warnings: 0
```

The default options for both `SELECT ... INTO OUTFILE` and `LOAD DATA INFILE` are quite reasonable and will work in most cases. There are two optional clauses `FIELDS` and `LINES` that can be used for specific cases where it is necessary to change the options such as quoting, field boundaries (to separate fields by a custom character such as the tab character or comma) and line boundaries.

For more information on the `FIELDS` and `LINES` options for both `LOAD DATA INFILE` and `SELECT ... INTO OUTFILE`, see the MySQL manual at `http://dev.mysql.com/doc/refman/6.0/en/load-data.html`.

- `LOAD XML INFILE` — The `LOAD XML INFILE` command can be used to load XML data into tables. The text file for input can be any XML file. To generate XML output by using the `mysql` client, use the `--xml` option, as shown here:

```
shell> mysql --xml -e 'SELECT * FROM sakila.film' > /tmp/film.xml
```

Remember, the output file does not contain the table structure! Use `mysqldump` to save the structure:

```
shell> mysqldump --no-data sakila film > /tmp/film-schema.sql
```

Here is a sample of the output generated by the command executed previously:

```
<?xml version="1.0"?>

<resultset statement="SELECT * FROM sakila.film
" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
 <field name="film_id">1</field>
 <field name="title">ACADEMY DINOSAUR</field>
 <field name="description">A Epic Drama of a Feminist And a Mad
Scientist who must Battle a Teacher in The Canadian Rockies</field>
 <field name="release_year">2006</field>
 <field name="language_id">1</field>
 <field name="original_language_id" xsi:nil="true" />
 <field name="rental_duration">6</field>
 <field name="rental_rate">0.99</field>
 <field name="length">86</field>
 <field name="replacement_cost">20.99</field>
 <field name="rating">PG</field>
 <field name="special_features">Deleted Scenes,Behind the
  Scenes</field>
 <field name="last_update">2006-02-15 05:03:42</field>
  </row>
```

The <row> and </row> tags are used to reference the start and end of a row in the output file. The <field name> and </field> tags are used to represent the columns in the row. The name attribute of the <field> tag specifies the name of the column.

In the following example the film table that was exported previously is loaded into an existing sakila2 database. First, the empty table with the proper schema must be created:

```
shell> mysql sakila2 < /tmp/film-schema.sql
```

Then, the data can be loaded with LOAD XML INFILE:

```
mysql> load xml infile '/tmp/film.xml' into table film;
Query OK, 1000 rows affected, 3 warnings (0.18 sec)
Records: 1000  Deleted: 0  Skipped: 0  Warnings: 3
```

The LOAD XML INFILE command was added in MySQL 6.0. More information about the available options for LOAD XML INFILE is available in the MySQL Manual at http://dev.mysql.com/doc/refman/6.0/en/load-xml.html.

DROP extensions

Similar to the IF NOT EXISTS extension to many CREATE statements, MySQL has the IF EXISTS extension to many DROP statements. For example:

```
mysql> DROP DATABASE IF EXISTS db_does_not_exist;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1008
Message: Can't drop database 'db_does_not_exist'; database
 doesn't exist
1 row in set (0.00 sec)
```

In addition to the IF EXISTS extension to many DROP statements, MySQL extends other DROP statements:

- DROP TABLE can delete one or more tables in a comma-separated list. For example:

```
mysql> use test;
Database changed
mysql> CREATE TABLE drop_me1 (id int);
Query OK, 0 rows affected (0.35 sec)

mysql> CREATE TABLE drop_me2 (id int);
Query OK, 0 rows affected (0.36 sec)

mysql> SHOW TABLES LIKE 'drop%';
```

```
+-----------------------+
| Tables_in_test (drop%) |
+-----------------------+
| drop_me1              |
| drop_me2              |
+-----------------------+
2 rows in set (0.00 sec)

mysql> DROP TABLE drop_me1, drop_me2;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW TABLES LIKE 'drop%';
Empty set (0.00 sec)
```

- Dropping an index with the `DROP INDEX` statement is nonstandard SQL. MySQL's `DROP INDEX` extension may take an `ONLINE` or `OFFLINE` option. Currently `DROP OFFLINE INDEX` has no function, as all `DROP INDEX` commands behave as if specified as `DROP ONLINE INDEX`.

The LIMIT extension

The `LIMIT` extension applies mostly to `SELECT` statements, although other statements may use the same syntax (such as `UPDATE`, `DELETE`, and `SHOW ERRORS`). It is a clause that begins with the reserved word `LIMIT` and takes one or two numeric arguments. If only one argument is present, it is the number of rows to constrain the output to. For example:

```
mysql> SELECT TABLE_SCHEMA, TABLE_NAME
    -> FROM INFORMATION_SCHEMA.TABLES
    -> WHERE ENGINE='InnoDB'
    -> LIMIT 5;
+--------------+------------+
| TABLE_SCHEMA | TABLE_NAME |
+--------------+------------+
| sakila       | actor      |
| sakila       | actor2     |
| sakila       | address    |
| sakila       | category   |
| sakila       | city       |
+--------------+------------+
5 rows in set (0.03 sec)
```

If the `LIMIT` clause has two arguments, the first value is the offset and the second value is the number of rows to constrain the output to. The offset starts at 0 (no offset) — thus, a single argument to `LIMIT` such as `LIMIT 5` acts as `LIMIT 0,5`. To get the middle three records from the previous example, use:

```
mysql> SELECT TABLE_SCHEMA, TABLE_NAME
    -> FROM INFORMATION_SCHEMA.TABLES
    -> WHERE ENGINE='InnoDB'
```

```
    -> LIMIT 1,3;
+--------------+------------+
| TABLE_SCHEMA | TABLE_NAME |
+--------------+------------+
| sakila       | actor2     |
| sakila       | address    |
| sakila       | category   |
+--------------+------------+
3 rows in set (0.03 sec)
```

The syntax for two arguments to LIMIT can be comma separated, as in the example above (LIMIT 1,3) or it can be specified as LIMIT 3 OFFSET 1.

Although the LIMIT clause can be useful, its implementation is very basic. In order to retrieve the information, mysqld processes a query as if there were no LIMIT, and stops when it reaches the row count it needs to. This means that a query, including an ORDER BY or GROUP BY with a LIMIT, still has to sort all the data. Additionally, a query that has a LIMIT and specifies an offset will have to process all the rows in the offset first — to retrieve the results of a query containing the clause LIMIT 99,20, the mysqld server will process 120 rows and return 20.

The LIMIT clause is the very last clause in a query or subquery.

SELECT extensions

The SELECT statement is one of the most frequently used SQL statements. In standard SQL, SELECT is a versatile tool for a wide variety of record retrieval and reporting activities. MySQL has extended the functionality of SELECT with many new nonstandard options and clauses, some of which relate to performance and backup.

ON the WEBSITE **MySQL has extended how the GROUP BY clause interacts with the SELECT fields by adding more aggregating functions, the WITH ROLLUP clause, ASC and DESC sort orders, and more. See the accompanying website for this book at www.wiley.com/go/mysqladminbible for explanations and examples of the GROUP BY extensions.**

The SELECT extensions SQL_CACHE and SQL_NO_CACHE control query interaction with the mysqld internal query cache. For information about the query cache and how to use these extensions, see Chapter 12.

SELECT . . . INTO OUTFILE/SELECT . . . INTO DUMPFILE

The SELECT...INTO OUTFILE command is used to create a text file of the contents of database table. This can be used to logically export an entire table or a subset of the table data. The mysqldump tool for logical export (See Chapter 13 for more information on mysqldump) can

support filters with its `--where` option; however it will always export all fields in a table. `SELECT...INTO OUTFILE` allows you to export only some fields.

By default, `SELECT...INTO OUTFILE` writes to a the file in `datadir`, but a location for the file can be specified optionally. The following shows how to export part of `sakila.rental`:

```
mysql> SELECT rental_id INTO OUTFILE '/tmp/rental-data.sql'
    -> FROM rental WHERE staff_id=1;
Query OK, 8042 rows affected (0.05 sec)
```

`SELECT...INTO OUTFILE` will not overwrite existing files. If the file specified already exists, `mysqld` throws an error:

```
ERROR 1086 (HY000): File '/tmp/rental-data.sql' already exists
```

There is no table definition included in the `SELECT...INTO OUTFILE` so you should make sure to save a copy of the table definition for restoration of the file.

The `SELECT INTO DUMPFILE` command works similarly to the `SELECT...INTO OUTFILE` command. However, it will only write one row with no processing of any kind. If you want to dump a BLOB object this would be a good option.

SQL_SMALL_RESULT/SQL_BIG_RESULT

With the `SELECT` statement the `SQL_SMALL_RESULT` option can be used in conjunction with the `GROUP BY` or `DISTINCT` clauses to specify that the result set of the query will be small enough that the server can use in-memory temporary tables. This could potentially result in faster execution.

The `SQL_BIG_RESULT` option is used in conjunction with the `GROUP BY` or `DISTINCT` clauses to specify that the result set of the query will be too large to fit an in-memory temporary table. Instead, a disk-based temporary table will be constructed.

UNION . . . ORDER BY

The `ORDER BY` clause can be used with the `UNION` statement joining two or more `SELECT` statements to specify a sort order to the returned results. Any column references in the `ORDER BY` clause are not allowed to include the table name. You should use an alias in the `SELECT` statement and then use this alias in the `ORDER BY` clause.

SELECT . . . FOR UPDATE

When using the `FOR UPDATE` clause a write lock is placed on any rows the `SELECT` statement processes. This lock is held for the duration of the transaction and released at the end of the transaction. For more information about transaction and locking, see Chapter 9.

SELECT . . . LOCK IN SHARE MODE

When using the LOCK IN SHARE MODE clause a read lock is placed on the rows the SELECT statement processes. Other transactions are allowed to read the locked rows, but they are not allowed to either update or delete any of the locked rows. This lock is released at the end of the transaction. See the "Row level lock" section of Chapter 9 for details on the LOCK IN SHARE MODE extension to SELECT.

DISTINCTROW

The DISTINCTROW option specifies that only distinct rows are returned in the result set of a SELECT statement. DISTINCTROW is a synonym of the SQL standard DISTINCT.

SQL_CALC_FOUND_ROWS

The SQL_CALC_FOUND_ROWS option is used to force mysqld to calculate how many rows are in the result set. After the SELECT with the SQL_CALC_FOUND_ROWS option finishes executing, the row count can be returned with the SELECT FOUND_ROWS() query. The following example demonstrates that using the LIMIT clause does not change the result of this calculation:

```
mysql> SELECT SQL_CALC_FOUND_ROWS rental_date, inventory_id,
    -> customer_id, return_date FROM RENTAL LIMIT 1\G
*************************** 1. row ***************************
rental_date: 2005-05-24 22:53:30
inventory_id: 367
 customer_id: 130
 return_date: 2005-05-26 22:04:30
1 row in set (0.01 sec)
```

In this case the LIMIT clause caused the SELECT to return data from one record. Now to see what the row count was:

```
mysql> SELECT FOUND_ROWS();
+--------------+
| found_rows() |
+--------------+
|        16044 |
+--------------+
1 row in set (0.00 sec)
```

Then to verify that the row count is accurate:

```
mysql> SELECT COUNT(*) FROM RENTAL;
+----------+
| count(*) |
+----------+
|    16044 |
+----------+
1 row in set (0.00 sec)
```

SQL_BUFFER_RESULT

Specifying `SQL_BUFFER_RESULT` in a `SELECT` means that the result sets of `SELECT` statements are placed into temporary tables. With storage engines that use table-level locking this can speed up the release of the table lock. There is a corresponding global system variable, `sql_buffer_result`, which controls this behavior for all `SELECT` statements. By default this system variable is set to `0` (off). Setting this system variable to `1` will enable it, and cause all `SELECT` statements to act as if they were `SELECT SQL_BUFFER_RESULT` statements.

HIGH_PRIORITY/LOW_PRIORITY

See the "Table-level locks" section in Chapter 9 for more information on using `SELECT HIGH_PRIORITY` and `SELECT LOW_PRIORITY` to change the behavior of how `mysqld` chooses the next lock to grant from the read and write lock queues.

- `DO` — Though not actually a `SELECT` extension, `DO` is a separate statement that can be used instead of `SELECT` to execute a statement and ignore the results. The syntax for `DO` is the same as for `SELECT`. Use `DO` when the query execution is the important part, not the results from the query execution (such as when running queries for the purpose of preloading the query cache). The `SLEEP()` function is a good example of a function whose execution is more important than its results:

```
mysql> SELECT SLEEP(1);
+----------+
| SLEEP(1) |
+----------+
|        0 |
+----------+
1 row in set (1.00 sec)

mysql> DO SLEEP(1);
Query OK, 0 rows affected (1.00 sec)
```

- `LIMIT` — See the section "The LIMIT extension" in this chapter for details.
- `PROCEDURE ANALYSE()` — See Chapter 5 for how to use `PROCEDURE ANALYSE()` to determine the optimal data type for fields already populated with data.
- `EXPLAIN SELECT` — See Chapter 18 for how to use `EXPLAIN SELECT` to analyze query performance.

Server maintenance extensions

MySQL has extended SQL to include server maintenance extensions. Most of these server maintenance extensions are described in other parts of this book; however, for the sake of completeness, they are listed here and the relevant chapter(s) are referenced.

All of the FLUSH statements are written to the binary log by default and will be replicated to any slaves. To change this default behavior, specify NO_WRITE_TO_BINLOG TABLE right after FLUSH, for example:

```
FLUSH NO_WRITE_TO_BINLOG TABLE TABLES;
```

TIP **LOCAL is a shorter alias for NO_WRITE_TO_BINLOG.**

The server maintenance statements are:

- KILL — KILL QUERY thread_id kills the query currently running from the thread_id thread. The values of thread_id for all connections to mysqld are shown in the output of SHOW PROCESSLIST and can be queried in the PROCESSLIST system view in the INFORMATION_SCHEMA database.
- KILL CONNECTION thread_id kills the query and the connection from the thread_id thread. KILL thread_id is an alias for KILL CONNECTION thread_id.

WARNING **The KILL CONNECTION and KILL QUERY statements both kill the query associated with the specified thread_id. However, if a connection is interrupted in any other way, the query will continue until it finishes or mysqld knows the connection has been broken. This means that pressing Ctrl-C to abort a long-running query may only abort the connection, not the query itself!**

It is important to always double-check that your expectations match reality. After using the KILL command, run a SHOW PROCESSLIST to ensure that the command is gone or has the status Killed, which means that mysqld is killing the process. After aborting a connection in any other way, reconnect to the database and check SHOW PROCESSLIST to make sure that there are no unwanted queries. This includes connections that were accidentally aborted, such as a network interruption, and programs aborted by external kill commands, such as Ctrl-C or an operating-system-level kill.

- FLUSH HOSTS, FLUSH TABLES, and FLUSH STATUS — These server maintenance extensions can be run as SQL statements in a client. They can also be run via the mysqladmin command line client, specifying flush-hosts, flush-tables, and flush-status. See the "mysqladmin" section of Chapter 3 for the description of what these statements do.
- FLUSH DES_KEY_FILE — Disregard the DES keys currently in memory and reload them from the file specified in the --des_key_file option to mysqld.
- FLUSH LOGS and FLUSH BACKUP LOGS — See Chapter 16 for more information about logs and the FLUSH LOGS and FLUSH BACKUP LOGS statements. FLUSH LOGS can also be run via mysqladmin; see the "mysqladmin" section of Chapter 3 for the description of what the flush-logs option does.
- FLUSH PRIVILEGES and FLUSH USER_RESOURCES — See Chapter 14 for more information about managing permissions and privileges, and the FLUSH PRIVILEGES and FLUSH USER_RESOURCES statements. FLUSH PRIVILEGES can also be run via mysqladmin; see the "mysqladmin" section of Chapter 3 for the description of what the flush-privileges option does.

- FLUSH TABLES WITH READ LOCK — This will lock the tables, preventing modifications from happening until the lock is released, flush MyISAM buffers to disk, and close any open file descriptors. The read lock can be released explicitly by issuing an UNLOCK TABLES command or by issuing a command that implicitly releases the lock.
- FLUSH QUERY CACHE and RESET QUERY CACHE — See Chapter 12 for the query cache and information about the FLUSH QUERY CACHE and RESET QUERY CACHE statements.
- RESET MASTER and RESET SLAVE — See Chapter 22 for information about how RESET MASTER and RESET SLAVE commands are used in replication setups.
- CACHE INDEX...IN — The CACHE INDEX statement is used to configure MyISAM tables to utilize a named key cache. The following command would configure table_one and table_two to use the key cache small_cache instead of the global key cache.

  ```
  mysql> CACHE INDEX table_one, table_two IN small_cache
  ```

  The named key cache must be created before the CACHE INDEX statement is run. To create a key cache called small_cache, you could include the following in your configuration file in the [mysqld] directive:

  ```
  small_cache.key_buffer_size=128M
  ```

- LOAD INDEX INTO CACHE — The LOAD INDEX INTO CACHE statement can be used to preload one ore more tables into a key cache. The key cache can be the default key cache or an explicitly named key cache. To preload the two tables used in the previous example:

  ```
  mysql> LOAD INDEX INTO CACHE table_one, table_two;
  ```

The SET extension and user-defined variables

The SET extension in mysqld is used to assign values to variables. Values can be assigned to user-defined variables, using either of the following syntaxes, which differ only in the assignment operator:

```
SET @varname:=value
SET @varname=value commands
```

In the first example, the assignment operator is := and the second syntax just uses = as the assignment operator. To use a user-defined variable, simply replace any number or string with the variable itself. For example:

```
mysql> SELECT 100+100;
+---------+
| 100+100 |
+---------+
|     200 |
+---------+
1 row in set (0.00 sec)

mysql> SET @num:=100;
Query OK, 0 rows affected (0.05 sec)
```

```
mysql> SELECT @num+100;
+----------+
| @num+100 |
+----------+
|      200 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT @num+@num;
+-----------+
| @num+@num |
+-----------+
|       200 |
+-----------+
1 row in set (0.00 sec)
```

Changing the value of a number is as easy as setting the value:

```
mysql> SET @num:=100+@num;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @num;
+------+
| @num |
+------+
|  200 |
+------+
1 row in set (0.00 sec)
```

User-defined variables are local in scope. They cannot be seen by other sessions, and if you exit the session, the user-defined variables are lost. User-defined variables are case-insensitive:

```
mysql> SELECT @NUM;
+------+
| @NUM |
+------+
|  200 |
+------+
1 row in set (0.01 sec)
```

In a `SELECT` statement, the `:=` assignment operator sets the value of a user-defined variable and returns the new value. For example:

```
mysql> SELECT @num, @num:=@num+100, @num;
+------+----------------+------+
| @num | @num:=@num+100 | @num |
+------+----------------+------+
|  200 |            300 |  300 |
+------+----------------+------+
```

```
1 row in set (0.01 sec)
mysql> SELECT @num, @num:=@num+100, @num;
+------+----------------+------+
| @num | @num:=@num+100 | @num |
+------+----------------+------+
|  300 |            400 |  400 |
+------+----------------+------+
1 row in set (0.00 sec)
```

Note how mysqld processes the query from left to right. This is an implementation detail that has been used for many purposes, including row numbering and running totals. For example, Ziesel wants to show a running total of rental fees and the average fee collected. She uses the payment table in the sakila database and two user-defined variables to keep track of the total count (@count) and the total amount of fees collected (@payments):

```
mysql> use sakila;
Database changed
mysql> SET @payments:=0, @count:=0;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @count:=@count+1 AS '#', amount,
    -> @payments:=@payments+amount AS running_total,
    -> @payments/@count AS running_avg
    -> FROM payment LIMIT 5;
+------+--------+---------------+-------------+
| #    | amount | running_total | running_avg |
+------+--------+---------------+-------------+
|    1 |   2.99 |          2.99 | 2.990000000 |
|    2 |   0.99 |          3.98 | 1.990000000 |
|    3 |   5.99 |          9.97 | 3.323333333 |
|    4 |   0.99 |         10.96 | 2.740000000 |
|    5 |   9.99 |         20.95 | 4.190000000 |
+------+--------+---------------+-------------+
5 rows in set (0.01 sec)
```

To be able to use the running average after the query is complete, Ziesel initializes a third variable, @run_avg, and changes the query to:

```
SELECT @count:=@count+1 AS '#', amount,
@payments:=@payments+amount AS running_total,
@run_avg:=@payments/@count AS running_avg
FROM payment LIMIT 5;
```

After the query is run, each variable retains its most current value. Ziesel can now use @run_avg in her next reporting query, if she so desires. Or, she can disconnect, and the values of @count, @payments and @run_avg will be NULL.

Local variables in stored code

Setting and manipulating local variables in stored code (such as stored procedures) is also done with SET and SELECT. However, in stored code, variables do not need @ in front of their names. See the sections on local variables in Chapter 7 for examples of how local variables are used in stored code.

Assigning values to dynamic server variables

Dynamic server variables can be changed while mysqld is running — there is no need to restart mysqld for the variable to be set. Server variables can be viewed at a GLOBAL or SESSION scope using SHOW GLOBAL VARIABLES and SHOW SESSION VARIABLES, respectively (see the SHOW extension later in this chapter). Similarly, dynamic server variables can be set on a GLOBAL or SESSION level as in the following:

```
mysql> SET GLOBAL max_allowed_packet=2*1024*1024;
Query OK, 0 rows affected (0.00 sec)

mysql> SET SESSION max_allowed_packet=4*1024*1024;
Query OK, 0 rows affected (0.00 sec)
```

Just as user-defined variables are accessible via a special prefix (@), server variables are similarly accessible, with the (@@) prefix:

```
mysql> SELECT @@global.max_allowed_packet,
    -> @@session.max_allowed_packet\G
*************************** 1. row ***************************
 @@global.max_allowed_packet: 2097152
@@session.max_allowed_packet: 4194304
1 row in set (0.00 sec)

mysql> SET @@session.max_allowed_packet = @@global.max_
 allowed_packet;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @@global.max_allowed_packet,  @@session.max_allowed_
 packet\G
*************************** 1. row ***************************
 @@global.max_allowed_packet: 2097152
@@session.max_allowed_packet: 2097152
1 row in set (0.00 sec)
```

TIP **As with SHOW VARIABLES, and SHOW STATUS, the SET server_variable command without a GLOBAL or SESSION scope setting will default to SESSION. To avoid confusion, always specify GLOBAL or SESSION. Similarly, always specify @@global.server_variable or @@session.server_variable in SELECT and SET statements.**

The LOCAL and @@local specifiers are aliases for SESSION and @@session, respectively. We recommend using SESSION and @@session so there is no question about the difference between a "local" server variable and a user-defined variable.

The SHOW extension

Metadata is available in the INFORMATION_SCHEMA database (See Chapter 21 for more details). Much of the information in the INFORMATION_SCHEMA database can be retrieved by using the SHOW extension. Although the SHOW syntax is less flexible than querying the INFORMATION_SCHEMA database, it is simpler than using a standard SQL query. SHOW statements are usually shorter than a standard SQL query, and thus faster to type. There are some SHOW commands that do not have INFORMATION_SCHEMA equivalents, such as the SHOW CREATE statements, which return CREATE statements.

The sql_quote_show_create system variable is a session-level variable settable via an option file such as my.cnf or via command line. This system variable takes a value of 0 or 1, with 1 being the default. When set to 0, identifiers (such as table, database, and field names) are not quoted:

```
mysql> select @@sql_quote_show_create;
+-------------------------+
| @@sql_quote_show_create |
+-------------------------+
|                       1 |
+-------------------------+
1 row in set (0.00 sec)

mysql> SHOW CREATE DATABASE sakila;
+----------+---------------------------------------------------+
| Database | Create Database                                   |
+----------+---------------------------------------------------+
| sakila   | CREATE DATABASE `sakila` /*!40100 DEFAULT
             CHARACTER SET latin1 */                           |
+----------+---------------------------------------------------+
1 row in set (0.41 sec)


mysql> set @@sql_quote_show_create=0;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW CREATE DATABASE sakila;
+----------+---------------------------------------------------+
| Database | Create Database                                   |
+----------+---------------------------------------------------+
| sakila   | CREATE DATABASE sakila /*!40100 DEFAULT
             CHARACTER SET latin1 */                           |
+----------+---------------------------------------------------+
1 row in set (0.00 sec)
```

Many SHOW commands support a LIKE clause, which will return all values where a specific field matches the pattern in the LIKE clause. For example, SHOW CHARACTER SET matches a LIKE pattern to the Charset field:

```
mysql> SHOW CHARACTER SET LIKE 'utf%';
+---------+---------------+--------------------+--------+
| Charset | Description   | Default collation  | Maxlen |
+---------+---------------+--------------------+--------+
| utf8mb3 | UTF-8 Unicode | utf8mb3_general_ci |      3 |
| utf8    | UTF-8 Unicode | utf8_general_ci    |      4 |
| utf16   | UTF-16 Unicode | utf16_general_ci  |      4 |
| utf32   | UTF-32 Unicode | utf32_general_ci  |      4 |
+---------+---------------+--------------------+--------+
4 rows in set (0.00 sec)
```

Some will also support a WHERE clause, which is more flexible than a LIKE clause:

```
mysql> SHOW CHARACTER SET WHERE Maxlen=4;
+---------+---------------+-------------------+--------+
| Charset | Description   | Default collation | Maxlen |
+---------+---------------+-------------------+--------+
| utf8    | UTF-8 Unicode | utf8_general_ci   |      4 |
| utf16   | UTF-16 Unicode | utf16_general_ci |      4 |
| utf32   | UTF-32 Unicode | utf32_general_ci |      4 |
+---------+---------------+-------------------+--------+
3 rows in set (0.00 sec)
```

These WHERE clauses can support multiple conditions:

```
mysql> SHOW CHARACTER SET WHERE Maxlen=4 AND Charset LIKE '%8';
+---------+--------------+-------------------+--------+
| Charset | Description  | Default collation | Maxlen |
+---------+--------------+-------------------+--------+
| utf8    | UTF-8 Unicode | utf8_general_ci  |      4 |
+---------+--------------+-------------------+--------+
1 row in set (0.00 sec)
```

The SHOW commands are:

- SHOW AUTHORS — Takes no input. Displays Name, Location and a Comment about the various authors of the MySQL codebase.
- SHOW BINLOG EVENTS — See “Replication and Logging,” Chapter 16.
- SHOW BINARY LOGS — See “Replication and Logging,” Chapter 16.
- SHOW CHARACTER SET — Displays the name (Charset), Description, Default collation and maximum number of bytes required to store one character (Maxlen) for the character sets supported by the mysqld server. This does not require input, although both LIKE and WHERE clauses are supported. LIKE matches against the Charset field.

- The CHARACTER_SETS system view in the INFORMATION_SCHEMA database contains the same information as the SHOW CHARACTER SET statement. The corresponding fields are CHARACTER_SET_NAME, DEFAULT_COLLATE_NAME, DESCRIPTION, and MAXLEN.
- SHOW COLLATION — Displays the name (Collation), character set (Charset), Id, whether or not it is the default collation for its character set (Default), whether it is compiled into the server (Compiled), and the amount of memory in bytes that is required to sort using this collation (Sortlen) . This does not require input, although both LIKE and WHERE clauses are supported. LIKE matches against the Collation field.
- The COLLATIONS system view in the INFORMATION_SCHEMA database contains the same information as the SHOW COLLATION statement. The corresponding fields are COLLATION_NAME, CHARACTER_SET_NAME, ID, IS_COMPILED, and IS_DEFAULT and SORTLEN.
- SHOW COLUMNS — See the information for the COLUMNS system view in Chapter 21, “MySQL Data Dictionary.”
- SHOW CONTRIBUTORS — Takes no input. Displays Name, Location, and a Comment about a few contributors to causes supported by the former company MySQL AB.
- SHOW COUNT(*) ERRORS — Displays the value of the error_count session variable:

  ```
  mysql> SHOW COUNT(*) ERRORS;
  +-----------------------+
  | @@session.error_count |
  +-----------------------+
  |                     0 |
  +-----------------------+
  1 row in set (0.00 sec)
  ```

  SHOW ERRORS provides more information about the error(s) from the previous command that generated errors. Supports the LIMIT clause (see the section “The LIMIT extension” earlier in this chapter).
- SHOW COUNT(*) WARNINGS — Displays the value of the warning_count session variable. SHOW WARNINGS provides more information about the error(s) from the previous command that generated errors, warnings or notes.
- SHOW CREATE DATABASE — Requires a database name as an input. Displays the name of the database (Database) and a CREATE statement that can be used to create the database (Create Database). For example:

  ```
  mysql> SHOW CREATE DATABASE sakila;
  +----------+-----------------------------------------------------+
  | Database | Create Database                                     |
  +----------+-----------------------------------------------------+
  | sakila   | CREATE DATABASE `sakila` /*!40100 DEFAULT
   CHARACTER SET latin1 */                                          |
  +----------+-----------------------------------------------------+
  1 row in set (0.41 sec)
  ```

A synonym for SHOW CREATE DATABASE is SHOW CREATE SCHEMA. See Chapter 21 for information on the SCHEMATA system view in the INFORMATION_SCHEMA database.

- SHOW CREATE EVENT — Requires an event name as an input. Displays the name of the event (Event), a CREATE statement that can be used to create the event (Create Event), the character set of the session in which the event was created (character_set_client), the collation of the session in which the event was created (collation_connection), and the collation of the database that the event is associated with (Database Collation). See Chapter 7 for more information on events, and Chapter 21 for information on the EVENTS system view in the INFORMATION_SCHEMA database.
- SHOW CREATE FUNCTION — Requires a function name as an input. Displays the name of the function (Function), a CREATE statement that can be used to create the function (Create Function), the character set of the session in which the function was created (character_set_client), the collation of the session in which the function was created (collation_connection), and the collation of the database that the function is associated with (Database Collation). See Chapter 7 for more information on stored functions, and Chapter 21 for information on the ROUTINES system view in the INFORMATION_SCHEMA database.
- SHOW CREATE PROCEDURE — Requires a procedure name as an input. Displays the name of the procedure (Procedure), a CREATE statement that can be used to create the procedure (Create Procedure), the character set of the session in which the procedure was created (character_set_client), the collation of the session in which the procedure was created (collation_connection), and the collation of the database that the procedure is associated with (Database Collation). See Chapter 7 for more information on stored procedures, and Chapter 21 for information on the ROUTINES system view in the INFORMATION_SCHEMA database.
- SHOW CREATE SCHEMA — See SHOW CREATE DATABASE.
- SHOW CREATE TABLE — Requires a table name as an input. Displays the name of the table (Table) and a CREATE statement that can be used to create the table (Create Table). See Chapter 21 for information on the TABLES system view in the INFORMATION_SCHEMA database.
- SHOW CREATE TRIGGER — Requires a trigger name as an input. Displays the name of the trigger (Trigger), the sql_mode of the session in which the trigger was created (sql_mode), a CREATE statement that can be used to create the trigger (SQL Original Statement), the character set of the session in which the trigger was created (character_set_client), the collation of the session in which the trigger was created (collation_connection), and the collation of the database that the trigger is associated with (Database Collation). See Chapter 7 for more information on triggers, and Chapter 21 for information on the TRIGGERS system view in the INFORMATION_SCHEMA database.

- SHOW CREATE VIEW — Requires a view name as an input. Displays the name of the view (View), a CREATE statement that can be used to create the view (Create View), the character set of the session in which the view was created (character_set_client), and the collation of the session in which the view was created (collation_connection). See Chapter 8 for more information on views, and Chapter 21 for information on the VIEWS system view in the INFORMATION_SCHEMA database.
- SHOW DATABASES — Displays the database name (Database). Does not require input, although both LIKE and WHERE clauses are supported. LIKE matches against the Database field.

  The SCHEMATA system view in the INFORMATION_SCHEMA database contains the same information as the SHOW DATABASES statement. The corresponding field is SCHEMA_NAME. The SCHEMATA system view also contains the DEFAULT_CHARACTER_SET_NAME and DEFAULT_COLLATION for the database, which the SHOW command does not contain.
- SHOW ENGINE — Requires an engine name and what type of information to see. Supported statements are:
  - SHOW ENGINE INNODB STATUS — Displays information about semaphores, foreign key errors, transactions, file I/O, the insert buffer, the adaptive hash index, logs, buffers and buffer pool, and row operations.
  - SHOW ENGINE INNODB MUTEX — Displays information about mutexes: Type (always Innodb), the source file where the mutex was created (Name) and Status, which contains a comma-separated set or subset of the following values:
  - count — How many times the mutex was requested.
  - spin_waits — How many times the spinlock ran.
  - spin_rounds — How many spinlock rounds.
  - os_waits — How many times the operating system had to wait due to a spinlock failing to acquire a mutex lock.
  - os_wait_times — If the timed_mutexes variable is set to 1, how much time, in ms, was spent on waiting for the operating system. This value is 0 if the timed_mutexes system variable is set to 0 or OFF, which it is by default.
  - os_yields — How many times the thread acquiring a mutex lock yielded to the operating system, giving up its time slice, in the hope that yielding will remove the barriers to acquiring the mutex lock.

    For example:

```
mysql> SHOW ENGINE INNODB MUTEX;
+--------+----------------------+-----------+
| Type   | Name                 | Status    |
+--------+----------------------+-----------+
| InnoDB | trx/trx0trx.c:143    | os_waits=0 |
| InnoDB | dict/dict0dict.c:1365 | os_waits=0 |
```

```
| InnoDB | dict/dict0mem.c:90    | os_waits=0 |
| InnoDB | dict/dict0dict.c:1365 | os_waits=0 |
| InnoDB | dict/dict0dict.c:1365 | os_waits=0 |
| InnoDB | dict/dict0dict.c:1365 | os_waits=0 |
| InnoDB | dict/dict0mem.c:90    | os_waits=0 |
 ...
```

Debugging InnoDB mutexes is beyond the scope of this book.

If mysqld supports the NDB cluster storage engine, SHOW ENGINE NDB STATUS and SHOW ENGINE NDBCLUSTER STATUS are supported. Either command will show information about the NDB storage engine.

- SHOW ENGINES — Takes no input. Displays information about storage engines, including name (Engine), how mysqld supports it (Support), Comment, and whether the storage engine supports transactions, XA, and savepoints. See Chapter 11 for more information on storage engines, and Chapter 21 for information on the ENGINES system view in the INFORMATION_SCHEMA database.

  Values for Support include DEFAULT (for the default storage engine), YES (for usable supported storage engines), and DISABLED (for supported storage engines that cannot be used). The NO value is not applicable, because storage engines can be runtime plugins.

- SHOW ERRORS — Displays the error number(s) and description(s) from the last command that generated an error. Supports the LIMIT clause (see the section "The LIMIT extension" earlier in this chapter).

- SHOW EVENTS — Displays the database the event is associated with (Db), Name, Definer, Time zone, Type (ONE TIME or RECURRING), Execute at (non-NULL for a ONE TIME event), Interval_value (non-NULL for a RECURRING event), Interval_Field (non-NULL for a RECURRING event), Starts (non-NULL for a RECURRING event), Ends (non-NULL for a RECURRING event), Status (ENABLED, DISABLED or SLAVESIDE_DISABLED), the server-id of the mysqld instance that created the event (Originator), the character set of the session in which the event was created (character_set_client), the collation of the session in which the event was created (collation_connection), and the collation of the database that the event is associated with (Database Collation).

  SHOW EVENTS does not require input. Without input, SHOW EVENTS will show all events associated with the current database. If there is no current database, error 1046 occurs:

  ```
  mysql> SHOW EVENTS;
  ERROR 1046 (3D000): No database selected
  ```

  To show events from a particular database, specify SHOW EVENTS FROM db_name. Both the LIKE and WHERE clauses are supported, and either can occur alone or with a FROM clause. LIKE matches against the Name field.

  The EVENTS system view in the INFORMATION_SCHEMA database contains the same information as the SHOW EVENTS statement. The corresponding fields are EVENT_SCHEMA, EVENT_NAME, DEFINER, TIME_ZONE, EVENT_TYPE, EXECUTE_AT. The EVENTS system view also contains the EVENT_BODY (always SQL), EVENT_DEFINITION, SQL_MODE,

ON_COMPLETION, CREATED, LAST_ALTERED, LAST_EXECUTED, and EVENT_COMMENT for the event, which the SHOW command does not contain. See Chapter 7 for more information on events, and Chapter 21 for more information about the EVENTS system view in the INFORMATION_SCHEMA database.

- SHOW FULL TABLES — See SHOW TABLES.
- SHOW FUNCTION CODE — Displays the ordinal position (Pos) and Instruction for each step in a stored function. This is only valid if mysqld was compiled with --with-debug:

```
mysql> SHOW FUNCTION CODE sakila.inventory_in_stock;
ERROR 1289 (HY000): The 'SHOW PROCEDURE|FUNCTION CODE'
feature is disabled; you need MySQL built with '--with-
debug' to have it working
```

  This is useful for debugging stored functions.

- SHOW FUNCTION STATUS — Displays the database the function is associated with (Db), Name, Type (FUNCTION), Definer, Modified, Created, Security_type (DEFINER or INVOKER), Comment, the character set of the session in which the event was created (character_set_client), the collation of the session in which the event was created (collation_connection), and the collation of the database that the event is associated with (Database Collation). See Chapter 7 for more information on stored functions, and Chapter 21 for information on the ROUTINES system view in the INFORMATION_SCHEMA database.

  Without input, SHOW FUNCTION STATUS will show all functions associated with all databases. Both the LIKE and WHERE clauses are supported, and either can occur alone or with a FROM clause. LIKE matches against the Name field.

- SHOW GRANTS — Displays the GRANT statement(s) that can be used to re-create the privileges for a particular user@host. With no input, SHOW GRANTS displays grant statements for the current user@host, which can be seen in the output of SELECT CURRENT_USER(). A different user@host, is specified with a FOR clause, for example:

```
SHOW GRANTS FOR guest@localhost;
```

- SHOW INDEX — Displays the index information for a table. For the meaning of the fields, see the information about the STATISTICS system view in the INFORMATION_SCHEMA database, in Chapter 21.

  A table name is required as part of a FROM clause. A database may be specified by using a second FROM clause. The following are all equivalent and will produce the output shown here:

```
SHOW INDEX FROM sakila.country\G
SHOW INDEX FROM country FROM sakila\G
USE sakila; SHOW INDEX FROM COUNTRY\G
*************************** 1. row ***************************
        Table: country
   Non_unique: 0
```

```
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: country_id
    Collation: A
  Cardinality: 109
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_Comment:
1 row in set (0.00 sec)
```

SHOW INDEXES and SHOW KEYS are aliases for SHOW INDEX.

- SHOW INDEXES — See SHOW INDEX.
- SHOW KEYS — See SHOW INDEX.
- SHOW MASTER STATUS — "See Replication and Logging" in Chapter 16.
- SHOW OPEN TABLES — Displays information about currently open tables. A table is open if a thread is using it, and the number of open tables depends on how many threads have opened a table. If two different threads use the same table, there are two open tables — because each thread opens the table. The table_open_cache server variable can be used to cache open tables to reduce the overhead of opening and closing the same table.
- SHOW OPEN TABLES — For each currently open non-temporary table, SHOW OPEN TABLES displays the Database, Table, whether or not the open table is being used (In_use), and whether the table name is locked (Name_locked, value is 1 when renaming and dropping tables). Without input, SHOW OPEN TABLES will show all open tables from all databases. To show open tables from a particular database, specify SHOW OPEN TABLES FROM db_name. Both the LIKE and WHERE clauses are supported, and either can occur alone or with a FROM clause. LIKE matches against the Table field:

```
mysql> SHOW OPEN TABLES from mysql LIKE 't%';
+----------+---------------------------+--------+-------------+
| Database | Table                     | In_use | Name_locked |
+----------+---------------------------+--------+-------------+
| mysql    | time_zone                 |      0 |           0 |
| mysql    | time_zone_name            |      0 |           0 |
| mysql    | time_zone_transition_type |      0 |           0 |
| mysql    | time_zone_leap_second     |      0 |           0 |
| mysql    | time_zone_transition      |      0 |           0 |
| mysql    | tables_priv               |      0 |           0 |
+----------+---------------------------+--------+-------------+
6 rows in set (0.00 sec)
```

- SHOW PLUGINS — Displays the Name, Status, Type, filename of the plugin (Library, NULL for built-in plugins such as storage engines) and License. For the meaning of the fields, see the information about the PLUGINS system view in the INFORMATION_SCHEMA database, in Chapter 21. The PLUGINS system view also contains additional information, including the PLUGIN_VERSION, PLUGIN_TYPE_VERSION (specifies which version of mysqld the plugin supports), IN_LIBRARY_VERSION, PLUGIN_AUTHOR, and PLUGIN_DESCRIPTION.
- SHOW PRIVILEGES — Display the name (Privilege), Context and Comment about each type of privilege that can be used in a GRANT statement. See the "Permissions" section of Chapter 21, "The MySQL Data Dictionary", for more information.
- SHOW PROCEDURE CODE — Displays the ordinal position (Pos) and Instruction for each step in a stored procedure. This is only valid if mysqld was compiled with --with-debug:

```
mysql> SHOW PROCEDURE CODE sakila.rewards_report;
ERROR 1289 (HY000): The 'SHOW PROCEDURE|FUNCTION CODE'
feature is disabled; you need MySQL built with '--with-
debug' to have it working
```

  This is useful for debugging stored procedures.
- SHOW PROCEDURE STATUS — Displays the database that the procedure is associated with (Db), Name, Type (PROCEDURE), Definer, Modified, Created, Security_type (DEFINER or INVOKER), Comment, the character set of the session in which the event was created (character_set_client), the collation of the session in which the event was created (collation_connection), and the collation of the database that the event is associated with (Database Collation). See Chapter 7 for more information on stored procedures, and Chapter 21 for information on the ROUTINES system view in the INFORMATION_SCHEMA database.
- Without input, SHOW PROCEDURE STATUS will show all procedures associated with all databases. Both the LIKE and WHERE clauses are supported, and either can occur alone or with a FROM clause. LIKE matches against the Name field.
- SHOW PROCESSLIST — See the information in Chapter 21, The MySQL Data Dictionary, for the PROCESSLIST system view in the INFORMATION_SCHEMA database.
- SHOW PROFILE — As of MySQL version 6.0.5, query profiling can be done on a session-level basis. By default, the profiling session variable is set to 0 and the PROFILING system view has no rows. If it is set to 1, queries can be profiled. The output of SHOW PROFILE is the Status and Duration of each step:

```
mysql> SHOW PROFILE;
Empty set (0.00 sec)

mysql> SET profiling=1;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT COUNT(*) FROM sakila.film;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)

mysql> SHOW PROFILE;
+--------------------+----------+
| Status             | Duration |
+--------------------+----------+
| starting           | 0.000126 |
| Opening tables     | 0.000057 |
| System lock        | 0.000016 |
| Table lock         | 0.000013 |
| init               | 0.000018 |
| optimizing         | 0.000009 |
| statistics         | 0.000018 |
| preparing          | 0.000014 |
| executing          | 0.000008 |
| Sending data       | 0.000634 |
| end                | 0.000010 |
| query end          | 0.000007 |
| freeing items      | 0.000054 |
| logging slow query | 0.000006 |
| cleaning up        | 0.000020 |
+--------------------+----------+
15 rows in set (0.00 sec)
```

This is very useful for debugging exactly why a query takes a long time.

Without input, `SHOW PROFILE` displays profiling information for the most recent query. With a `FOR QUERY n` clause, profiling information will be shown for the query with a `Query_ID` of `n`. The `Query_ID` value is obtained from the `SHOW PROFILES` statement. `SHOW PROFILE` accepts the `LIMIT` clause (see The LIMIT Extension earlier in this chapter).

The `SHOW PROFILE` statement also accepts an optional comma-separated list of extra information to show in addition to `Status` and `Duration`. Table 4-4 shows the values and the information returned (for the description of the fields, please refer to the information in Chapter 21 about the `PROFILING` system view.

To show partial output with the CPU and Source information for our sample query (`SELECT COUNT(*) FROM sakila.film`):

```
mysql> SHOW PROFILE CPU, SOURCE FOR QUERY 1 LIMIT 2 OFFSET 0\G
*************************** 1. row ***************************
         Status: starting
       Duration: 0.000126
```

```
       CPU_user: 0.000109
     CPU_system: 0.000016
Source_function: NULL
    Source_file: NULL
    Source_line: NULL
*************************** 2. row ***************************
         Status: Opening tables
       Duration: 0.000057
       CPU_user: 0.000033
     CPU_system: 0.000023
Source_function: open_tables
    Source_file: sql_base.cc
    Source_line: 3588
2 rows in set (0.00 sec)
```

TABLE 4-4

SHOW PROFILE Extra Field Information

Extra Name in SHOW PROFILE Statement	Fields Shown
ALL	Status, Duration, CPU_user, CPU_system, Context_voluntary, Context_involuntary, Block_ops_in, Block_ops_out, Messages_sent, Messages_received, Page_faults_major, Page_faults_minor, Swaps, Source_function, Source_file, Source_line
BLOCK IO	Status, Duration, Block_ops_in, Block_ops_out
CONTEXT SWITCHES	Status, Duration, Context_voluntary, Context_involuntary
CPU	Status, Duration, CPU_user, CPU_system
IPC	Status, Duration, Messages_sent, Messages_received
MEMORY	Status, Duration
PAGE FAULTS	Status, Duration, Page_faults_major, Page_faults_minor
SOURCE	Status, Duration, Source_function, Source_file, Source_line
SWAPS	Status, Duration, Swaps

- SHOW PROFILES — As of MySQL version 6.0.5, query profiling can be done on a session-level basis. By default, the profiling session variable is set to 0 and the

PROFILING system view has no rows. If it is set to 1, queries can be profiled. The SHOW PROFILES statement is related to, but very different from, the SHOW PROFILE statement. SHOW PROFILES outputs profiling information for the most recent queries. There is one query per row, and the maximum number of queries shown is determined by the profiling_history_size session variable. This session variable also restricts the number of queries that are saved in the PROFILING system view.

```
mysql> SHOW PROFILES;
+----------+------------+---------------------------------+
| Query_ID | Duration   | Query                           |
+----------+------------+---------------------------------+
|        1 | 0.00101000 | SELECT COUNT(*) FROM sakila.film |
+----------+------------+---------------------------------+
1 row in set (0.00 sec)
```

- SHOW SCHEMAS — See SHOW DATABASES.
- SHOW SLAVE HOSTS — See “Replication and Logging,” Chapter 16.
- SHOW SLAVE STATUS — See “Replication and Logging,” Chapter 16.
- SHOW STATUS — See the information in Chapter 21 on the GLOBAL_STATUS and SESSION_STATUS system views in the INFORMATION_SCHEMA database.

WARNING SHOW STATUS **shows session variables by default. To avoid confusion, always specify either** SHOW GLOBAL STATUS **or** SHOW SESSION STATUS**.**

- SHOW TABLE STATUS — See the information in Chapter 21 on the TABLES system view in the INFORMATION_SCHEMA database.
- SHOW STORAGE ENGINES — See SHOW ENGINES.
- SHOW TABLES — Displays table names in an output field beginning with Tables_in_. Without input, SHOW TABLES will show all tables and views associated with the current database. If there is no current database, error 1046 occurs:

```
mysql> SHOW TABLES;
ERROR 1046 (3D000): No database selected
```

To show tables and views from a particular database, specify SHOW TABLES FROM db_name. Both the LIKE and WHERE clauses are supported, and either can occur alone or with a FROM clause. LIKE matches against the Tables_in_ field:

```
mysql> SHOW TABLES LIKE 'f%';
+----------------------+
| Tables_in_sakila (f%) |
+----------------------+
| film                 |
| film_actor           |
| film_category        |
| film_list            |
| film_text            |
+----------------------+
5 rows in set (0.00 sec)
```

SHOW FULL TABLES adds one field to the information in SHOW TABLES. Table_type, specifies whether the table is a base table, view or system view:

```
mysql> SHOW FULL TABLES LIKE 'f%';
+-----------------------+------------+
| Tables_in_sakila (f%) | Table_type |
+-----------------------+------------+
| film                  | BASE TABLE |
| film_actor            | BASE TABLE |
| film_category         | BASE TABLE |
| film_list             | VIEW       |
| film_text             | BASE TABLE |
+-----------------------+------------+
5 rows in set (0.01 sec)
```

Only the INFORMATION_SCHEMA views are system views. See Chapter 8 for more information on views, and Chapter 21 for detailed information about the INFORMATION_SCHEMA database.

- SHOW TRIGGERS — See the information in Chapter 21 on the TRIGGERS system view in the INFORMATION_SCHEMA database.
- SHOW VARIABLES—See the information in Chapter 21 on the GLOBAL_VARIABLES and SESSION_VARIABLES system views in the INFORMATION_SCHEMA database.

WARNING SHOW VARIABLES **shows session variables by default. To avoid confusion, always specify either** SHOW GLOBAL VARIABLES **or** SHOW SESSION VARIABLES**.**

- SHOW WARNINGS — Displays the error, warning and note number(s) and description(s) from the last command that generated an error, warning, or note. Supports the LIMIT clause (see "The LIMIT extension" earlier in this chapter).

Table definition extensions

MySQL has added features to tables, which require extended SQL syntax in order to specify. The following SQL extensions are related to nonstandard table features in MySQL:

- AUTO_INCREMENT=num — Set the AUTO_INCREMENT value for the table. See Chapter 5 for more details on AUTO_INCREMENT.
- AVG_ROW_LENGTH=num — Set the average row length. This helps mysqld allocate proper space for records and is only applicable for very large MyISAM tables (over 256 Tb). See the MySQL manual at http://dev.mysql.com/doc/refman/6.0/en/create-table.html for more details on when and how to set this parameter, and how other variables interact.
- CHARACTER SET charset_name — Specify the default character set for fields created in this table. See the "Character sets and collations" section earlier in this chapter for more information.
- CHECKSUM=1 — This will enable a *live* checksum of a table. Every time a record is changed, the checksum is updated. Tables with CHECKSUM=1 keep the current value of CHECKSUM TABLE tblname stored in their metadata, using very little overhead. Setting

CHECKSUM=0 will disable this feature. This feature only works on MyISAM tables and is set to 0 by default.

- COLLATE collation_name — Specify the default collation for fields created in this table. See the "Character sets and collations" section earlier in this chapter for more information.
- COMMENT='comment string' — Give the table a descriptive comment, that can be seen in the TABLES system view of the INFORMATION_SCHEMA database and in the output of SHOW TABLE STATUS and SHOW CREATE TABLE.

  Fields and indexes can also be given a comment within a definition in a CREATE TABLE or ALTER TABLE statement. The COMMENT 'comment string' syntax gives the field or index a descriptive comment that can be seen in the COLUMNS (for fields) or STATISTICS (for indexes) system views of the INFORMATION_SCHEMA database and in the output of SHOW CREATE TABLE. Note that unlike the COMMENT option for a table, for fields and indexes the COMMENT keyword is separated from the comment string by a space, not an equals sign (=).
- CONNECTION='connection string' — For tables using the FEDERATED storage engine, this specifies the connection information. See Chapter 11 for more information on the FEDERATED storage engine and how to specify a connection string.
- DATA DIRECTORY='path_to_dir' — MyISAM tables will store their .MYD files in the path_to_dir directory. This option is ignored for ALTER TABLE statements but is honored in most CREATE TABLE statements — the exception is when a table is created with partitioning, the table-level DATA DIRECTORY option is ignored.
- DELAY_KEY_WRITE=1 — This will delay index buffer updates until a table is closed and then update an index buffer all at once. This makes writes faster, as index updates are batched, but can lead to corruption. DELAY_KEY_WRITE=0 will flush the index buffer every time the index is updated. This option is only for MyISAM tables. In addition, this option depends on the value of the system variable delay_key_write. By default, delay_key_write is set to ON, which means that MyISAM tables default to a DELAY_KEY_WRITE value of 0. If the delay_key_write server variable is set to OFF, no MyISAM tables delay key writes even if DELAY_KEY_WRITE is set to 1. If the delay_key_write server variable is set to ALL, the default setting for DELAY_KEY_WRITE is 1.
- ENGINE=storage_engine_name — This sets the storage engine of the table. For more information about storage engines, see Chapter 11.
- INDEX DIRECTORY='path_to_dir' — MyISAM tables will store their .MYI files in the path_to_dir directory. This option is ignored for ALTER TABLE statements but is honored in most CREATE TABLE statements — the exception is when a table is created with partitioning, the table-level INDEX DIRECTORY option is ignored.
- INSERT_METHOD=method_name — Sets how an INSERT to a MERGE table should behave. If method_name is set to FIRST, records will be inserted into the first table in the UNION definition. If method_name is set to LAST, records will be inserted into the last

table in the UNION definition. If method_name is set to NO, the table will be marked as read-only and an insert into the merge table will result in an error:

```
ERROR 1036 (HY000): Table 'merge_test' is read only
```

- KEY_BLOCK_SIZE=num — The value of num is given to the storage engine as a suggested index block size. The storage engine may or may not use the value of num. The default is 0, which indicates that the storage engine should use its default index block size.
- MAX_ROWS=num — This helps mysqld allocate proper space for records and is only applicable for very large MyISAM tables (over 256 Tb). See the MySQL manual at http://dev.mysql.com/doc/refman/6.0/en/create-table.html for more details on when and how to set this parameter, and how other variables interact.
- MIN_ROWS=num — The minimum number of rows expected to be stored in this table.
- PACK_KEYS=value — The default value is DEFAULT, which specifies that for MyISAM tables, long indexes for CHAR, VARCHAR, BINARY, and VARBINARY fields are compressed. If value is set to 1, number fields are also compressed. If value is set to 0, no indexes are compressed.

  Partitioning-related extensions include defining partitions and subpartitions with PARTITION BY, and changing partitioning with ADD PARTITION, DROP PARTITION, COALESCE PARTITION, REORGANIZE PARTITION, ANALYZE PARTITION, CHECK PARTITION, OPTIMIZE PARTITION, REBUILD PARTITION, REPAIR PARTITION, and REMOVE PARTITIONING. These are discussed in Chapter 15.
- PASSWORD='password_string' — This option provides no functionality.
- ROW_FORMAT='row_format_name' — Currently there are six different row format types. Each storage engine uses one or more of the following row format types:
  - Default — Uses the default row format for the storage engine.
  - Compact — The default InnoDB row format. Uses more processing power, storing less data.
  - Redundant — An older InnoDB row format. Requires less processing power but stores redundant data.
  - Fixed — Rows are a fixed-width, minimizing fragmentation. Fixed is the default for MyISAM, Falcon, BLACKHOLE, CSV, and MEMORY storage engines, and is used unless the table contains a variable-width field.
  - Dynamic — Rows are variable-width, containing one or more variable-width fields, such as VARCHAR, TEXT or BLOB. Used by the MyISAM, Falcon, BLACKHOLE, and CSV storage engines.
  - Page — The Maria storage engine uses the Page row format by default. No other storage engine uses this row format.
  - Compressed — Rows are compressed and read-only. Compressed is the default and only row format for the ARCHIVE storage engine and MyISAM tables when compressed with myisampack.

- UNION=(tbl_list) — In a MERGE table, the UNION clause specifies a comma-separated list of tables that the MERGE table is a wrapper for. The first and last table names in this list are the tables used when INSERT_METHOD is set to FIRST and LAST, respectively. See Chapter 11 for more information on MERGE tables.

Table maintenance extensions

Indexes stay up to date with regard to the data within the index data structure. However, indexes require periodic maintenance for stability, speed and metadata updates. Table maintenance solves the following problems:

- Out-of-date table structure
- Index and data corruption
- Index and data fragmentation
- Out-of-date index and data statistics

MySQL has several commands to maintain index and table data:

- CHECK TABLE
- REPAIR TABLE
- CHECKSUM TABLE
- ANALYZE TABLE
- OPTIMIZE TABLE

Index and data corruption

Corruption can occur to both the data and the indexes belonging to a table, and may occur for several reasons. The most common cause of corruption is when data and index files are changed at the file system level, or when mysqld crashes, such as when there is not enough RAM or the host machine is turned off abruptly without shutting down mysqld properly. Other, more infrequent causes of table corruption are hardware problems, such as a malfunctioning RAID controller or corrupted RAM, and bugs in the client code, mysqld code, or storage engine code.

To determine if a table has corruption, use the CHECK TABLE command:

```
mysql> USE sakila;
Database changed
mysql> CHECK TABLE film\G
*************************** 1. row ***************************
   Table: sakila.film
      Op: check
Msg_type: status
Msg_text: OK
1 row in set (0.02 sec)
```

CHECK TABLE is only supported by tables using the MyISAM, InnoDB, ARCHIVE, and CSV storage engines. If you try to use CHECK TABLE on a table that does not support it, the Msg_text field of the output contains:

```
The storage engine for the table doesn't support check
```

Other storage engines such as Falcon, PBXT, and Maria implement their own methods of checking for table corruption and performing repairs; See Chapter 11 for more details. When table corruption does occur, the output from CHECK TABLE will include a Msg_type of error and the Msg_text field will describe the problem:

```
mysql> CHECK TABLE film_text\G
*************************** 1. row ***************************
   Table: sakila.film_text
      Op: check
Msg_type: error
Msg_text: Unexpected byte: 5 at link: 1065187756
*************************** 2. row ***************************
   Table: sakila.film_text
      Op: check
Msg_type: error
Msg_text: Corrupt
2 rows in set (54.18 sec)
```

CHECK TABLE takes a comma-separated list of one or more tables and supports the following options:

- EXTENDED — As the name implies, this takes a longer time to run than any other option. However, this option will perform a full lookup on all keys and indexes, checking for 100% data consistency.
- MEDIUM — This is the default option used if no option is specified. For every table, calculate a checksum for the indexes on each data row, comparing the final result to the checksum of the index rows. Also verify that deleted links are valid.
- CHANGED — Only check a table if it was changed since the last time it was checked, or if the table was not closed properly. If a table is checked, the checks that are done are the same as the MEDIUM option.
- FAST — Only check a table if it was not closed properly. If a table is checked, the checks that are done are the same as the MEDIUM option.
- QUICK — Calculate a checksum for the indexes on each data row, comparing the final result to the checksum of the index rows. Same as MEDIUM, without the verification for deleted links.
- FOR UPGRADE — Checks to see if the table is out of date due to a server upgrade. Although this is a quick check, if the table is found to be out of date, a MEDIUM check will be run automatically, which can take some time.

Options are specified after the list of tables:

```
mysql> CHECK TABLE film_text, film FAST;
+-----------------+-------+----------+---------------------------+
| Table           | Op    | Msg_type | Msg_text                  |
+-----------------+-------+----------+---------------------------+
| sakila.film_text | check | status   | Table is already up to date |
| sakila.film     | check | status   | OK                        |
+-----------------+-------+----------+---------------------------+
2 rows in set (0.01 sec)
```

Other common warnings and errors are described in the following list. Note that these are actual errors we have encountered, but the table names have been changed to `sakila.film_text` to protect privacy:

- `Table './sakila/film_text' is marked as crashed and should be repaired`
- `1 client is using or hasn't closed the table properly`
- `Incorrect information in file: './sakila/film_text.frm'`
- `Table './sakila/film_text' is marked as crashed and last (automatic?) repair failed`
- `Invalid key block position: 284290829344833891 key block size: 1024 file_length: 4453643264`
- `key delete-link-chain corrupted`
- `Table 'sakila.film_text' doesn't exist`
- `Record-count is not ok; is 330426316 Should be: 330426389`
- `Size of datafile is: 0 Should be: 172`
- `Found 1533 deleted space. Should be 0`
- `Found 73 deleted blocks Should be: 0`

If `CHECK TABLE` returns a `Msg_type` of `error`, you will need to attempt to fix the problem. The first step to try when fixing a corrupt table is `REPAIR TABLE`. Only the MyISAM, ARCHIVE, and CSV storage engines support `REPAIR TABLE`, and you must have the `INSERT` and `SELECT` privileges on the tables you want to repair. The following example shows a successful `REPAIR TABLE` command, followed by a `CHECK TABLE` command as a sanity check to ensure that the table is actually not corrupt anymore:

```
mysql> REPAIR TABLE film_text\G
*************************** 1. row ***************************
   Table: sakila.film_text
      Op: repair
Msg_type: warning
Msg_text: Number of rows changed from 4733691 to 4733690
*************************** 2. row ***************************
   Table: sakila.film_text
      Op: repair
```

```
Msg_type: status
Msg_text: OK
2 rows in set (5 min 24.40 sec)

mysql> CHECK TABLE film_text\G
*************************** 1. row ***************************
   Table: sakila.film_text
      Op: check
Msg_type: status
Msg_text: OK
1 row in set (36.08 sec)
```

Note that the REPAIR TABLE in this case took almost five and a half minutes. REPAIR TABLE makes all the data and indexes for that table unavailable for the duration of the repair. Canceling the repair before it is complete is a bad idea as it will only add to the current corruption.

REPAIR TABLE optionally takes one of three options after the table name is specified:

- QUICK — Only a repair of the index tree is attempted.
- EXTENDED — Instead of attempting to fix indexes by doing a REPAIR BY SORT on one entire index at a time, the index is rebuilt one row at a time.
- USE_FRM — Uses the .frm file to rebuild the index, disregarding the existing .MYI index file. This option should be used only as a last resort, as the .MYI file has important information that will be lost, such as the AUTO_INCREMENT value. Also, using USE_FRM can cause fragmentation in the table records.

NOTE **If a table needs to be repaired because of a mysqld upgrade, do not use the USE_FRM option. Before mysqld version 6.0.6, the table may be truncated, removing of all data, if the USE_FRM option was used when the table needed to be repaired because of an upgrade. In versions 6.0.6 and higher, attempting to specify USE_FRM when a table needed to be repaired because of an upgrade returns a Msg_type of error and a Msg_text of Failed repairing incompatible .FRM file. In this situation, try a simple REPAIR TABLE tblname with no options first.**

Should you need to use USE_FRM, a successful REPAIR TABLE will likely return at least two rows, one of which issues a warning that the Number of rows changed from 0 to a greater number. This is because at the start of the REPAIR TABLE, the existing .MYI file was disregarded, so the number of rows in the index at the start of the repair process was 0. Sample output is shown here:

```
mysql> REPAIR TABLE film_text USE_FRM\G
*************************** 1. row ***************************
   Table: sakila.film_text
      Op: repair
Msg_type: warning
Msg_text: Number of rows changed from 0 to 1000
*************************** 2. row ***************************
```

```
    Table: sakila.film_text
       Op: repair
 Msg_type: status
 Msg_text: OK
2 rows in set (0.08 sec)
```

There are some command-line tools available that can be used for table repair. For example the Maria storage engine has command-line tool called maria_chk that can be used to check, repair, and optimize Maria tables. MyISAM can be managed with the myisamchk utility. Both of these tools must be used while mysqld is shut down, or you risk causing further corruption by changing the files on the file system when mysqld is still using them.

For tables created using storage engines that do not support the REPAIR TABLE command, you can try rebuilding the table using an ALTER TABLE table_name ENGINE = storage_engine command, which will force a rebuild of the data and indexes. Like REPAIR TABLE, rebuilding the data and indexes with this type of ALTER TABLE command will make all the data and indexes for the table unusable for the duration of the rebuild. Again, canceling the ALTER TABLE before it is complete may add to the current corruption.

TIP **It is best to fix table corruption when the table does not need to be used immediately. Chapter 22 has scaling and high availability architectures that you may want to implement. If table corruption happens on one of a few slave servers, the slave server can be taken out of production while the corruption is fixed. In the meantime, the other slave servers can split the extra load. If the table corruption happens on a master server, but the slave servers have no corruption, promote a slave to be the new master (as discussed in Chapter 22), and take the old master out of production service while the corruption is being fixed. In this way, taking a database offline for maintenance does not require noticeable downtime for your application. This tip is extremely useful for proactive maintenance such as upgrading mysqld.**

Using the methods previously outlined should resolve any problems if the table corruption is in the index data. If the indexes are still corrupt, or if the corruption is in the row data itself, your only choice may be to restore a previous version from backup. Backups and restoration of data are covered in Chapter 13.

Fragmentation

Fragmentation of the data and indexes can occur when the ordering of the index pages on the disk are not similar to the index ordering of the records on the pages. Fragmentation also occurs when there are a large number of unused pages in the blocks allocated for the index. Fragmentation is most often caused when data is deleted, leaving gaps in the index and data files that may not be filled even when a new row is inserted.

Resolving fragmentation can be difficult. With some storage engines such as MyISAM you can use the OPTIMIZE TABLE command. This will resolve data and index fragmentation issues. As with a REPAIR TABLE command, the data and indexes will be unavailable for the duration of the OPTIMIZE TABLE.

Here is an example of a successful OPTIMIZE TABLE command:

```
mysql> OPTIMIZE TABLE film_text;
+------------------+----------+----------+----------+
| Table            | Op       | Msg_type | Msg_text |
+------------------+----------+----------+----------+
| sakila.film_text | optimize | status   | OK       |
+------------------+----------+----------+----------+
1 row in set (0.11 sec)
```

Tables using the InnoDB storage engine map the OPTIMIZE TABLE command to an ALTER TABLE command. While this will defragment the row data, it will not always defragment the index data. If the index data is not defragmented by an OPTIMIZE TABLE, only a logical data export and reimport will resolve index fragmentation. See Chapter 13 for how to export and reimport data.

Reclaiming disk space

When a table that stores its data and indexes directly on the file system is defragmented, the size of the files decrease. For example, a fragmented MyISAM table will have smaller .MYD and .MYI files after defragmentation. The disk space is automatically reclaimed.

When InnoDB is used and mysqld is set to use innodb_file_per_table, table data and indexes are stored in a .ibd file in the data directory. All table metadata is stored together in a centralized ibdata file. When an InnoDB table is defragmented, its .ibd file will shrink and disk space will automatically be reclaimed.

However, by default, mysqld is *not* set to use innodb_file_per_table, and InnoDB puts all of the metadata, data and indexes for all tables into a centralized ibdata file. When an InnoDB table is defragmented on this configuration, the ibdata file will not shrink, even though the data is successfully defragmented. The good news is that the space is not lost — InnoDB will add that space to its pool of free space and put new rows in it. The amount of InnoDB free space reported in the TABLE_COMMENT field of the INFORMATION_SCHEMA.TABLES system view and the Comment field of SHOW TABLE STATUS for an InnoDB table will increase. The bad news is that the operating system cannot reclaim that disk space.

TIP **If the size of your data is more than a few hundred gigabytes, consider utilizing** innodb_file_per_table **so that defragmentation can reclaim disk space. Many organizations actually see a performance improvement when switching to** innodb_file_per_table **because writes and reads are happening from several different** .ibd **files instead of one or two centralized** ibdata **files!**

Maintaining table statistics

The maintenance of data and indexes should include maintaining the metadata that the server stores about table statistics. This is important because the query optimizer uses this information

in choosing which indexes, if any, to use when executing a query. To recalculate statistics, use the ANALYZE TABLE command as shown here:

```
mysql> ANALYZE TABLE film;
+-------------+---------+----------+----------+
| Table       | Op      | Msg_type | Msg_text |
+-------------+---------+----------+----------+
| sakila.film | analyze | status   | OK       |
+-------------+---------+----------+----------+
1 row in set (0.15 sec)
```

ANALYZE, REPAIR, and OPTIMIZE TABLE statements are written to the binary log by default, and will be replicated to any slaves. To change this default behavior, specify NO_WRITE_TO_BINLOG TABLE between the first word and the word TABLE — for example:

```
REPAIR NO_WRITE_TO_BINLOG TABLE film;
```

LOCAL is a shorter alias for NO_WRITE_TO_BINLOG and can be used with any of the three statements.

Getting a table checksum

To get a checksum of the entire table, use the CHECKSUM TABLE tblname command. By default, mysqld will return a live checksum if this is supported by the table (see the "Table definition extensions" section earlier in this chapter for more information). If the table does not support a live checksum, mysqld will calculate a checksum of the table. This requires a full table scan and can be very slow.

CHECKSUM TABLE tblname optionally takes one of two options at the end of the statement:

- QUICK — Returns the live checksum if supported by the table, otherwise returns NULL.
- EXTENDED — Calculates a checksum of the table, even if live checksum is supported.

Transactional statement extensions

In MySQL, several statements cause an *implicit transaction* — that is, they will perform an *implicit commit* before and after executing the actual statement. The commits, and thus the transaction caused by a COMMIT before and a COMMIT after the statement, are done without your approval and without informing you. These commands include commands that start transactions, change the mysql system database, DDL (Data Definition Language) commands that change the schema and some server maintenance commands:

- ANALYZE TABLE
- ALTER -- DATABASE, EVENT, FUNCTION, PROCEDURE, TABLE, VIEW
- BACKUP DATABASE
- BEGIN, BEGIN WORK

- CACHE INDEX
- CHECK TABLE
- CREATE -- DATABASE, EVENT, FUNCTION, INDEX, PROCEDURE, TABLE, TRIGGER, USER, VIEW
- DROP -- DATABASE, EVENT, FUNCTION, INDEX, PROCEDURE, TABLE, TRIGGER, USER, VIEW
- FLUSH
- GRANT
- LOAD INDEX INTO CACHE
- LOCK TABLES
- RENAME -- TABLE, USER
- OPTIMIZE TABLE
- REPAIR TABLE
- RESTORE
- REVOKE
- SET -- PASSWORD, autocommit=1
- START TRANSACTION
- TRUNCATE TABLE
- UNLOCK TABLES

Implicit commits are not performed if there is no change. For example, SET autocommit=1 only performs an implicit commit if the value was previously 0. Implicit commits are also not performed before or after the ALTER TEMPORARY TABLE, CREATE TEMPORARY TABLE, and DROP TEMPORARY TABLE statements.

MySQL extends the SQL standard transactional statements with the following:

- By default, mysqld runs in *autocommit* mode. This means that every SQL statement is its own transaction, and an atomic transaction with more than one SQL statement does not occur. The system variable autocommit controls autocommit mode, and is set to 1 (ON) by default. To use transactions, SET autocommit=0 in the client or start mysqld with the autocommit system variable set to 0 (OFF).
- Alternatively, explicitly starting a transaction with START TRANSACTION will turn off autocommit mode for the duration of the transaction.
- START TRANSACTION can be specified with an optional WITH CONSISTENT SNAPSHOT statement. This will attempt, but not guarantee, to make transactions have consistent reads. If the transactional tables used in the transaction are InnoDB, a consistent snapshot will occur if the isolation level is REPEATABLE READ or SERIALIZABLE. The default isolation level for mysqld is REPEATABLE READ; for more information about transactions and isolation levels, see Chapter 9.

- When a transaction completes with either `COMMIT` or `ROLLBACK`, the default `mysqld` behavior (specified by a `completion_type` system variable with a value of `0`) is to finish working with the transaction but not begin a new transaction. This default behavior can be changed in a few ways:
  - To immediately start a new transaction, specify `COMMIT CHAIN` or `ROLLBACK CHAIN`. This will *chain* transactions, so they occur back to back without needing to explicitly start a new transaction when the previous transaction is finished. To change the default `mysqld` behavior to always chain transactions, set the `completion_type` server variable to `1`.
  - To have `mysqld` disconnect the client immediately after a transaction completes, specify `COMMIT RELEASE` or `ROLLBACK RELEASE`. This will *release* the client connection after the transaction is finished. To change the default `mysqld` behavior to always release connections after a transaction completes, set the `completion_type` server variable to `2`.
  - To override the behavior of the `completion_type` server variable when it is set to `1` or `2`, specify `NO RELEASE` or `NO CHAIN` after `COMMIT` or `ROLLBACK`. For example, if the `completion_type` server variable was set to `1` (always chain transactions), specifying `ROLLBACK NO CHAIN` would override the chain behavior specified by `completion_type`.

Summary

This chapter has shown the nonstandard SQL that MySQL supports, and the ways in which MySQL deviates from the SQL standard. You should have learned the following topics from this chapter:

- MySQL language structure
- MySQL deviations
- DML and `SELECT` command extensions
- The `SET` extension
- Table maintenance extensions

Contents

Introduction xxvii

Part I First Steps with MySQL

Chapter 1: Introduction to MySQL 3

MySQL Mission — Speed, Reliability, and Ease of Use 3

Company background 4

Community and Enterprise server versions 5

The MySQL Community 6

How to contribute 6

Reasons to contribute 7

Summary 7

Chapter 2: Installing and Upgrading MySQL Server 9

Before Installation 9

Choosing the MySQL version 11

MySQL support 12

Downloads 12

Installation 12

MySQL Server installations on Unix 13

MySQL Server Installation on Windows 20

Installing MySQL from a Noinstall Zip Archive 24

Starting and stopping MySQL from the Windows command line 25

Starting and stopping MySQL as a Windows service 26

Initial Configuration 29

Unix configuration file 31

Windows configuration file 31

MySQL Configuration Wizard on Windows 31

Detailed Configuration 32

The Server Type screen 33

Database Usage screen 33

InnoDB Tablespace screen 34

Concurrent Connections screen 34

Networking Options and Strict Mode Options screen 34

Character Set screen 35

Service Options screen 35

Security Options screen 35

Confirmation screen 36

MySQL Post-Install Configuration on Unix ... **36**
Initializing the system tables ... 36
Setting initial passwords ... 37
Root user password assignment ... 37
Anonymous users ... 39
Securing Your System ... **40**
Windows PATH Variable Configuration ... **42**
Automated startup ... 42
Starting and stopping mysqld on System V-based Unix ... 42
System V run levels ... 43
Upgrading mysqld ... **45**
The MySQL changelog ... 45
Upgrading MySQL on Windows ... 46
Troubleshooting ... **47**
Summary ... **48**

Chapter 3: Accessing MySQL . . . 49

Accessing mysqld with Command-Line Tools ... **49**
Frequently used options ... 50
Using the command-line mysql client ... 52
mysqladmin — Client for administering a server ... 62
GUI Tools ... **66**
SQLyog ... 66
phpMyAdmin ... 69
MySQL Query Browser ... 71
MySQL Administrator ... 74
MySQL Workbench ... 80
Summary ... **83**

Part II Developing with MySQL

Chapter 4: How MySQL Extends and Deviates from SQL . . . 87

Learning MySQL Language Structure ... **88**
Comments and portability ... 88
Case-sensitivity ... 90
Escape characters ... 91
Naming limitations and quoting ... 93
Dot notation ... 95
Time zones ... 97
Character sets and collations ... 98
Understanding MySQL Deviations ... **105**
Privileges and permissions ... 110
Transaction management ... 110
Check constraints ... 111
Upsert statements ... 112

Using MySQL Extensions **114**
Aliases 115
ALTER TABLE extensions 115
CREATE extensions 118
DML extensions 119
DROP extensions 124
The LIMIT extension 125
SELECT extensions 126
SELECT . . . INTO OUTFILE/SELECT . . . INTO DUMPFILE 126
SQL_SMALL_RESULT/SQL_BIG_RESULT 127
UNION . . . ORDER BY 127
SELECT . . . FOR UPDATE 127
SELECT . . . LOCK IN SHARE MODE 128
DISTINCTROW 128
SQL_BUFFER_RESULT 129
HIGH_PRIORITY/LOW_PRIORITY 129
Server maintenance extensions 129
The SET extension and user-defined variables 131
The SHOW extension 135
Table definition extensions 147
Table maintenance extensions 150
Transactional statement extensions 156
Summary **158**

Chapter 5: MySQL Data Types 159

Looking at MySQL Data Types **159**
Character String Types **160**
Length 162
Character string type attributes 164
National Character String Types **166**
Binary Large Object String Types **168**
BLOB values 169
BINARY values 169
BINARY length 169
VARBINARY length 170
Numeric Types **170**
Numeric data sizes and ranges 172
Numeric data type attributes 177
Boolean Types **180**
Datetime Types **183**
Allowed input values 185
Microsecond input 186
Automatic updates 187
Conversion issues 188
Numeric functions and DATETIME types 188

Other conversion issues 190
Datetime data type attributes 191
The effect of time zones 192
Interval Types 193
ENUM and SET Types 195
Enumerations 195
ENUM and SET data type attributes 198
Choosing SQL Modes 201
Invalid data 201
SQL modes 203
Using NULL Values 211
Finding an Optimal Data Type for Existing Data 212
Small data samples and PROCEDURE ANALYSE() 215
Summary 217

Chapter 6: MySQL Index Types 219

Looking at Keys and Indexes 219
Using Indexes to Speed Up Lookups 221
Creating and dropping indexes 223
Index order 225
Index length 226
Index types 228
Redundant indexes 230
Creating and Dropping Key Constraints 231
Creating and dropping unique key constraints 231
Creating and dropping foreign key constraints 232
Foreign key constraints and data changes 234
Requirements for foreign key constraints 235
Using FULLTEXT Indexes 237
Summary 239

Chapter 7: Stored Routines, Triggers, and Events 241

Comparing Stored Routines, Triggers, and Events 241
Using Triggers 242
Creating a trigger 243
Dropping a trigger 244
Multiple SQL statements in triggers 245
Changing a trigger 246
Triggers on views and temporary tables 247
Trigger runtime behavior 248
Finding all triggers 252
Trigger storage and backup 252
Triggers and replication 254
Trigger limitations 254

Using Stored Routines 255
Performance implications of stored routines 256
Stored procedures vs. stored functions 256
Creating a stored routine 256
Invoking a stored procedure 259
Dropping a stored routine 261
Multiple SQL statements in stored routines 261
INOUT arguments to a stored procedure 261
Local variables 262
Stored routine runtime behavior 264
Options when creating routines 265
Creating a basic stored function 268
Full CREATE FUNCTION syntax 269
Invoking a stored function 269
Changing a stored routine 270
Naming: stored routines 271
Stored procedure result sets 273
Stored routine errors and warnings 274
Conditions and handlers 275
Stored routine flow control 282
Recursion 284
Stored routines and replication 285
Stored function limitations 285
Stored routine backup and storage 286
Using Cursors 287
Using Events 289
Turning on the event scheduler 289
Creating an event 291
Dropping an event 292
Multiple SQL statements in events 293
Start and end times for periodic events 293
Event status 294
Finding all events 295
Changing an event 295
After the last execution of an event 296
Event logging 297
Event runtime behavior 298
Event limitations 299
Event backup and storage 300
Summary 300

Chapter 8: MySQL Views 301

Defining Views 302
View definition limitations and unexpected behavior 304
Security and privacy 305

Specify a view's definer 306
Abstraction and simplification 307
Performance 308
Updatable views 313
Changing a View Definition 317
Replication and Views 317
Summary 318

Chapter 9: Transactions in MySQL 319

Understanding ACID Compliance 320
Atomicity 321
Consistency 321
Isolation 321
Durability 321
Using Transactional Statements 322
BEGIN, BEGIN WORK, and START TRANSACTION 322
COMMIT 322
ROLLBACK 322
Savepoints 323
AUTOCOMMIT 324
Using Isolation Levels 325
READ UNCOMMITED 329
READ COMMITTED 331
REPEATABLE READ 332
SERIALIZABLE 334
Multi-version concurrency control 335
Explaining Locking and Deadlocks 336
Table-level locks 338
Page-level locks 341
Row-level locks 341
Recovering MySQL Transactions 343
Summary 344

Part III Core MySQL Administration

Chapter 10: MySQL Server Tuning 349

Choosing Optimal Hardware 349
Tuning the Operating System 352
Operating system architecture 352
File systems and partitions 353
Buffers 356
Kernel parameters 357
Linux 357
Other daemons 360

Tuning MySQL Server **360**
Status variables 360
System variables 361
Option file 361
Dynamic variables 371
Summary **373**

Chapter 11: Storage Engines 375

Understanding Storage Engines **375**
Storage engines as plugins 376
Storage engine comparison 376
Using Different Storage Engines **378**
MyISAM storage engine 378
InnoDB storage engine 384
MEMORY storage engine 394
Maria storage engine 396
Falcon storage engine 401
PBXT storage engine 410
FEDERATED storage engine 415
NDB storage engine 417
Archive storage engine 417
Blackhole storage engine 419
CSV storage engine 420
Working with Storage Engines **421**
CREATE TABLE 421
ALTER TABLE 421
DROP TABLE 422
Summary **422**

Chapter 12: Caching with MySQL 423

Implementing Cache Tables **424**
Working with the Query Cache **427**
What gets stored in the query cache? 427
Query cache memory usage and tuning 429
Query cache fragmentation 433
Utilizing memcached **434**
Summary **438**

Chapter 13: Backups and Recovery 439

Backing Up MySQL **439**
Uses for backups 441
Backup frequency 443
What to back up 445
Backup locations 445
Backup methods 445

Online backup 460
mysqlhotcopy 462
Commercial options 464
Copying Databases to Another Machine 467
Recovering from Crashes 468
Planning for Disasters 471
Summary 472

Chapter 14: User Management 473

Learning about MySQL Users 473
Access Control Lists 474
Wildcards 475
System tables 476
Managing User Accounts 478
GRANT and REVOKE commands 481
SHOW GRANTS and mk-show-grants 485
Resetting the Root Password 487
Windows server 488
Unix-based server 489
Debugging User Account Problems 490
Bad password 490
Access issues 491
Client does not support authentication protocol 491
Can't connect to local mysqld through socket '/path/to/mysqld.sock' 492
I do not have the right permissions! 493
Summary 494

Chapter 15: Partitioning 495

Learning about Partitioning 495
Partitioning Tables 496
RANGE partitioning 497
LIST partitioning 502
HASH partitioning 503
KEY partitioning 504
Composite partitioning 504
Partition management commands 507
Restrictions of partitioning 510
MERGE Tables 510
Creating a MERGE table 511
Changing a MERGE table 512
Advantages of MERGE tables 513
Partitioning with MySQL Cluster 513
Programmatic Partitioning 514
Summary 514

Chapter 16: Logging and Replication 517
Log Files 517
Error log 517
Binary logs 518
Relay logs 520
General and slow query logs 520
Rotating logs 522
Other methods of rotating 523
Replication 524
Setting up semisynchronous replication 525
Statement-based, row-based, and mixed-based replication 527
Replication Configurations 529
Simple replication 529
CHANGE MASTER statement 534
More complex setups 534
Additional replication configuration options 539
Correcting Data Drift 540
mk-table-checksum overview 540
mk-table-sync overview 542
Putting this together 542
Summary 543
Chapter 17: Measuring Performance 545
Benchmarking 546
mysqlslap 547
SysBench 552
Benchmarking recommendations 565
Profiling 566
SHOW GLOBAL STATUS 566
mysqltuner 568
mysqlreport 572
mk-query-profiler 580
mysqldumpslow 583
Capacity Planning 585
Summary 585
Part IV Extending Your Skills
Chapter 18: Query Analysis and Index Tuning 589
Using EXPLAIN 590
EXPLAIN plan basics 590
Data access strategy 596
EXPLAIN plan indexes 606
Rows 607

Extra ... 608
Subqueries and EXPLAIN ... 611
EXPLAIN EXTENDED ... 612
EXPLAIN on Non-SELECT Statements ... 614
Other Query Analysis Tools ... 614
Optimizing Queries ... 615
Factors affecting key usage ... 615
Optimizer hints ... 616
Adding an Index ... 616
Optimizing away Using temporary ... 620
Using an index by eliminating functions ... 623
Non-index schema changes ... 626
Batching expensive operations ... 628
Optimizing frequent operations ... 629
Summary ... 631

Chapter 19: Monitoring Your Systems . . . 633

Deciding What to Monitor ... 634
Examining Open Source Monitoring ... 636
Nagios ... 636
Cacti ... 637
Hyperic HQ ... 638
OpenNMS ... 640
Zenoss Core ... 641
Munin ... 642
Monit ... 643
Examining Commercial Monitoring ... 644
MySQL enterprise monitor ... 644
MONyog ... 645
Summary ... 646

Chapter 20: Securing MySQL . . . 649

Access Control Lists ... 649
Wildcards and blank values ... 650
Privilege and privilege levels ... 651
Accessing the Operating System ... 654
Database access ... 654
Changing MySQL connectivity defaults ... 654
Operating system login ... 654
Securing Backups and Logs ... 656
Data Security ... 656
Data flow ... 657
Encrypted connectivity ... 659
Data security using MySQL objects ... 664

Creating Security Policies **665**
Summary **666**

Chapter 21: The MySQL Data Dictionary 667

Object Catalog **668**
SCHEMATA 668
TABLES 670
VIEWS 674
COLUMNS 676
STATISTICS 679
TABLE_CONSTRAINTS 681
KEY_COLUMN_USAGE 682
REFERENTIAL_CONSTRAINTS 684
TRIGGERS 685
ROUTINES 686
PARAMETERS 690
EVENTS 691
PARTITIONS 693
System Information **695**
CHARACTER_SETS 695
COLLATIONS 696
COLLATION_CHARACTER_SET_APPLICABILITY 696
ENGINES 697
PLUGINS 697
PROCESSLIST 698
PROFILING 709
GLOBAL_VARIABLES 710
SESSION_VARIABLES 710
GLOBAL_STATUS 711
SESSION_STATUS 711
Displaying Permissions **711**
COLUMN_PRIVILEGES 712
TABLE_PRIVILEGES 713
SCHEMA_PRIVILEGES 714
USER_PRIVILEGES 715
Storage Engine-Specific Metadata **716**
Custom Metadata **716**
Defining the plugin 716
Compiling the plugin 722
Installing the plugin 724
Summary **725**

Chapter 22: Scaling and High Availability Architectures 727

Replication **728**
One read slave 729
Promoting a new master 729

Many read slaves ... 734
Master/master replication ... 735
Circular replication ... 736
SAN ... 737
DRBD ... 738
MySQL and DRBD setup ... 738
MySQL Proxy ... 739
Scaling read queries ... 740
Automated failover ... 740
Read/write splitting ... 742
Sharding ... 742
Linux-HA Heartbeat ... 742
MySQL Cluster ... 744
Connection Pooling ... 746
memcached ... 747
Summary ... 748

Appendix A: MySQL Proxy . . . 749

Appendix B: Functions and Operators . . . 783

Appendix C: Resources . . . 813

Index . . . 821