The MySQL Test Framework, Version 2.0

Abstract

This manual describes the MySQL test framework, consisting of the test driver and the test script language.

This manual is no longer updated. Any further updates to test framework documention take place in the MySQL Source Code documentation and can be accessed at The MySQL Test Framework, Version 2.0.

For help with using MySQL, please visit either the MySQL Forums or MySQL Mailing Lists, where you can discuss your issues with other MySQL users.

For additional documentation on MySQL products, including translations of the documentation into other languages, and downloadable versions in variety of formats, including HTML and PDF formats, see the MySQL Documentation Library.

Document generated on: 2017-07-20 (revision: 53027)

Table of Contents

Preface and Legal Notices

MySQL distributions include a set of test cases and programs for running them. These tools constitute the MySQL test framework that provides a means for verifying that MySQL Server and its client programs operate according to expectations. The test cases consist mostly of SQL statements, but can also use test language constructs that control how to run tests and verify their results.

This manual describes the MySQL test framework. It describes the programs used to run tests and the language used to write test cases.

Much of the content of this manual is based on material originally written by (in alphabetic order) Omer BarNir, Kent Boortz, and Matthias Leich. Updates and adaptations to version 2 and documentation of new features were done by Bjorn Munch.

Legal Notices

Chapter 1 Introduction to the MySQL Test Framework

MySQL distributions include a test suite: a set of test cases and programs for running them. (If you find that the test suite is not included in your distribution, look for a similar distribution with `-test` in the name and install that as well.) These tools constitute the MySQL test framework that provides a means for verifying that MySQL Server and its client programs operate according to expectations. The test cases consist mostly of SQL statements, but can also use test language constructs that control how to run tests and verify their results. Distributions also provide facilities for running unit tests and creating new unit tests.

This document describes the components of the MySQL test framework, how the test programs work, and the language used for writing test cases. It also provides a tutorial for developing test cases and executing them.

What is described here is version 2 of the test framework, which replaced version 1 from MySQL version 5.1.32. Version 1 is no longer supported and may eventually be removed.

Any feature described here as being available from MySQL 5.5.X is also available in any MySQL 5.6 or higher since the official release of 5.6 (version 5.6.10). It may not be available in all pre-GA releases of 5.6 that have been made available for download. Features described as availbale from MySQL 5.6 or 5.6.X are not available in any 5.5 release (unless otherwise stated) and may not be available in pre-GA releases of 5.6.

Similarly, features described as available in 5.7 are at least available in 5.7.7 and will be in the first GA release of 5.7. Unless otherwise stated, those features are not available in 5.6.

The application that runs the test suite is named `mysql-test-run.pl`. Its location is the `mysql-test` directory, which is present both in source and binary MySQL Server distributions.

On platforms other than Windows, `mysql-test-run.pl`is also available through the shortened name `mtr` in the same directory, as either a symbolic link or a copy.

The `mysql-test-run.pl` application starts MySQL servers, restarts them as necessary when a specific test case needs different start arguments, and presents the test result. For each test case, `mysql-test-run.pl` invokes the `mysqltest` program (also referred to as the “test engine”) to read the test case file, intepret the test language constructs, and send SQL statements to the server.

Input for each test case is stored in a file, and the expected result from running the test is stored in another file. The actual result is compared to the expected result after running the test.

For a MySQL source distribution, `mysql-test-run.pl` is located in the `mysql-test` directory, and `mysqltest` is located in the `client` directory. The `mysql-test` and `client` directories are located in the root directory of the distribution.

For a MySQL binary distribution, `mysql-test-run.pl` is located in the `mysql-test` directory, and `mysqltest` is located in the same directory where other client programs such as `mysql` or `mysqladmin` are installed. The locations of the `mysql-test` and other directories depend on the layout used for the distribution format.

Within the `mysql-test` directory, test case input files and result files are stored in the `t` and `r` directories, respectively. The input and result files have the same basename, which is the test name, but have extensions of `.test` and `.result`, respectively. For example, for a test named “decimal,” the input and result files are `mysql-test/t/decimal.test` and `mysql-test/r/decimal.result`.

Each test file is referred to as one test case, but usually consists of a sequence of related tests. An unexpected failure of a single statement in a test case makes the test case fail.

There are several ways a test case can fail:

- The `mysqltest` test engine checks the result codes from executing each SQL statement in the test input. If the failure is unexpected, the test case fails.
- A test case can fail if an error was expected but did not occur (for example, if an SQL statement succeeded when it should have failed).
- The test case can fail by producing incorrect output. As a test runs, it produces output (the results from `SELECT`, `SHOW`, and other statements). This output is compared to the expected result found in the `mysql-test/r` directory (in a file with a `.result` suffix). If the expected and actual results differ, the test case fails. The actual test result is written to a file in the `mysql-test/r` directory with a `.reject` suffix, and the difference between the `.result` and `.reject` files is presented for evaluation.
- A test case will fail if the MySQL server dies unexpectedly during the test. If this happens, the `mysqltest` test client will usually also report a failure due to loosing the connection.
- Finally, the test case will fail if the error log written by the MySQL server during the test includes warnings or errors which are not filtered (suppressed). See Section 4.16, “Suppressing Errors and Warning” for more about suppressing warnings.

This method of checking test results puts some restrictions on how test cases can be written. For example, the result cannot contain information that varies from run to run, such as the current time. However, if the information that varies is unimportant for test evaluation, there are ways to instruct the test engine to replace those fields in the output with fixed values.

Because the test cases consist mostly of SQL statements in a text file, there is no direct support for test cases that are written in C, Java, or other languages. Such tests are not within the scope of this test framework. But the framework does support executing your own scripts and initiating them with your own data. Also, a test case can execute an external program, so in some respects the test framework can be extended for uses other than testing SQL statements. Finally, it is possible to embed small pieces of Perl code within the test; this can sometimes be used to perform actions or execute logic which is beyond the capabilities of the test language or SQL.

Chapter 2 MySQL Test Framework Components

Table of Contents

The MySQL test framework consists of programs that run tests, and directories and files used by those programs.

Test Framework Programs

The MySQL test framework uses several programs:

- The `mysql-test-run.pl` Perl script is the main application used to run the test suite. It invokes `mysqltest` to run individual test cases.
- `mysqltest` runs test cases. Prior to MySQL 8.0, a version named `mysqltest_embedded` is available; it is similar to `mysqltest` but is built with support for the `libmysqld` embedded server.
- The `mysql_client_test` program is used for testing aspects of the MySQL client API that cannot be tested using `mysqltest` and its test language. Prior to MySQL 8.0, a version named `mysql_client_test_embedded` is available; it is similar to `mysql_client_test` but is used for testing the embedded server.
- The `mysql-stress-test.pl` Perl script performs stress-testing of the MySQL server.
- A unit-testing facility is provided so that individual unit test programs can be created for storage engines and plugins.

Test suite programs can be found in these locations:

- For a source distribution, `mysqltest` is in the `client` directory. For a binary distribution, it is in the MySQL `bin` directory.
- For a source distribution, `mysql_client_test` is in the `tests` directory. For a binary distribution, it is in the MySQL `bin` directory.
- The other programs are located in the `mysql-test` directory. For a source distribution, `mysql-test` is found under the source tree root. For a binary distribution, the location of `mysql-test` depends on the layout used for the distribution format.

Test Framework Directories and Files

The test suite is located in the `mysql-test` directory, which contains the following components:

- The `mysql-test-run.pl` and `mysql-stress-test.pl` programs that are used for running tests.
- The `t` directory contains test case input files. A test case file might also have additional files associated with it.
  - A file name of the form *`test_name`*`.test` is a test case file for a test named *`test_name`*. For example, `subquery.test` is the test case file for the test named `subquery`.

- A file name of the form *test_name*-master.opt provides options to associate with the named test case. mysql-test-run.pl restarts the server with the options given in the file if the options are different from those required for the currently running server.

  Note that the -master.opt file is used for the “main” server of a test, even if no replication is involved.

- A file name of the form *test_name*-slave.opt provides slave options.

- In MySQL 8.0.1, server options which need to be passed during initialization of the server can be passed in both *test_name*-master.opt and *test_name*-slave.opt. Each bootstrap variable must be passed as the value of a --bootstrap option so that mysql-test-run.pl can recognize that the variable should be used during server initialization, remove the existing data directory, and initialize a new data directory with the options provided. mysql-test-run.pl will also start the server afterwards, with the bootstrap options, and other options specified.

- A file name of the form *test_name*.cnf contains additional entries for the config file to be used for this test.

- A file name of the form *test_name*-master.sh is a shell script that will be executed before the server is started for the named test case. There may also be a *test_name*-slave.sh which is executed before the slave server is started.

  These files will not work on Windows and may therefore be replaced with a more portable mechanism in the future.

- The disabled.def file contains information about deferred/disabled tests. When a test is failing because of a bug in the server and you want it to be ignored by mysql-test-run.pl, list the test in this file.

  The format of a line in the disabled.def file looks like this, where fields are separated by one or more spaces or Tab characters:

  ```
  test_name : BUG#nnnnn YYYY-MM-DD disabler comment
  ```

  Example:

  ```
  rpl_row_blob_innodb : BUG#18980 2006-04-10 kent  Test fails randomly
  ```

  test_name is the test case name. BUG#*nnnnn* indicates the bug related to the test that causes it to fail (and thus requires it to be disabled). *disabler* is the name of the person that disabled the test. *comment* normally provides a reason why the test was disabled.

  The text following the colon is not part of the mandatory syntax for this file; mysql-test-run.pl will not actually care what is written here. However, it is a recommended standard.

  MySQL 5.5.17 introduces an additional element: after the test name and before the colon you may add one or more @*platform* to have the test disabled only on specific platform(s). Basic OS names as reported by $^O in Perl, or 'windows' are supported. Alternatively, @!*platform* will disable the test on all except the named platform.

  A comment line can be written in the file by beginning the line with a “#” character.

- The r directory contains test case result files:

- A file name of the form *`test_name`*`.result` is the expected result for the named test case. A file `r/`*`test_name`*`.result` is the output that corresponds to the input in the test case file `t/`*`test_name`*`.test`.
- A file name of the form *`test_name`*`.reject` contains output for the named test case if the test fails due to output mismatch (but not it it fails for other reasons).

For a test case that succeeds, the `.result` file represents both the expected and actual result. For a test case that fails, the `.result` file represents the expected result, and the `.reject` file represents the actual result.

If a `.reject` file is created because a test fails, `mysql-test-run.pl` removes the file later the next time the test succeeds.

- The `include` directory contains files that are included by test case files using the `source` command. These include files encapsulate operations of varying complexity into a single file so that you can perform the operations in a single step. See Section 4.12, “Using Include Files to Simplify Test Cases”.
- The `lib` directory contains library files used by `mysql-test-run.pl`, and database initialization SQL code.
- The `std_data` directory contains data files used by some of the tests.
- The `var` directory is used during test runs for various kinds of files: log files, temporary files, trace files, Unix socket files for the servers started during the tests, and so forth. This directory cannot be shared by simultaneous test runs.
- The `suite` directory contains a set of subdirectories containing named test suites. Each subdirectory represents a test suite with the same name.
- The `collections` directory contains collections of test runs that are run during integration and release testing of MySQL. They are not directly useful outside this context, but need to be part of the source repository and are included for reference.

Unit test-related files are located in the `unittest` directory. Additional files specific to storage engines and plugins may be present under the subdirectories of the `storage` or `plugin` directories.

Test Execution and Evaluation

There are a number of targets in the top-level `Makefile` that can be used to run sets of tests. `make test` runs all the tests. Other targets run subsets of the tests, or run tests with specific options for the test programs. Have a look at the `Makefile` to see what targets are available.

A “test case” is a single file. The case might contain multiple individual test commands. If any individual command fails, the entire test case is considered to fail. Note that “fail” means “does not produce the expected result.” It does *not* necessarily mean “executes without error,” because some tests are written precisely to verify that an illegal statement does in fact produce an error. In such an instance, if the statement executes successfully without producing the expected error, that is considered failure of the test.

Test case output (the test result) consists of:

- Input SQL statements and their output. Each statement is written to the result followed by its output. Columns in output resulting from SQL statements are separated by tab characters.
- The result from `mysqltest` commands such as `echo` and `exec`. The commands themselves are not echoed to the result, only their output.

The `disable_query_log` and `enable_query_log` commands control logging of input SQL statements. The `disable_result_log` and `enable_result_log` commands control logging of SQL statement results, and warning or error messages resulting from those statements.

`mysqltest` reads a test case file from its standard input by default. The `--test-file` or `-x` option can be given to name a test case file explicitly.

`mysqltest` writes test case output to the standard output by default. The `--result-file` or `-R` option can be used to indicate the location of the result file. That option, together with the `--record` option, determine how `mysqltest` treats the test actual and expected results for a test case:

- If the test produces no results, `mysqltest` exits with an error message to that effect, unless `--result-file` is given and this file is empty.
- Otherwise, if `--result-file` is not given, `mysqltest` sends test results to the standard output.
- With `--result-file` but not `--record`, `mysqltest` reads the expected results from the given file and compares them with the actual results. If the results do not match, `mysqltest` writes a `.reject` file in the same directory as the result file and exits with an error. It will also print out a diff of the expected and actual result, provided it can find an appropriate `diff` tool to use.
- With both `--result-file` and `--record`, `mysqltest` updates the given file by writing the actual test results to it. The file does not need to pre-exist.

`mysqltest` itself knows nothing of the `t` and `r` directories under the `mysql-test` directory. The use of files in those directories is a convention that is used by `mysql-test-run.pl`, which invokes `mysqltest` with the appropriate options for each test case to tell `mysqltest` where to read input and write output.

2.1 Test Framework System Requirements

The `mysqltest` and `mysql_client_test` programs are written in C++ and are available on any system where MySQL itself can be compiled, or for which a binary MySQL distribution is available.

Other parts of the test framework such as `mysql-test-run.pl` are Perl scripts and should run on systems with Perl installed.

`mysqltest` uses the `diff` program to compare expected and actual test results. If `diff` is not found, `mysqltest` writes an error message and dumps the entire contents of the `.result` and `.reject` files so that you can try to determine why a test did not succeed. If your system does not have `diff`, you may be able to obtain it from one of these sites:

```
http://www.gnu.org/software/diffutils/diffutils.html
http://gnuwin32.sourceforge.net/packages/diffutils.htm
```

`mysql-test-run.pl` cannot function properly if started from within a directory whose full path includes a space character, due to the complexities of handling this correctly in all the different contexts it will be used.

2.2 The Test Framework and SSL

When `mysql-test-run.pl` starts, it checks whether `mysqld` supports SSL connections:

- If `mysqld` supports SSL, `mysql-test-run.pl` starts it with the proper `--ssl-xxx` options that enable it to accept SSL connections for those test cases that require secure connections (those with “ssl” in their name). As `mysql-test-run.pl` runs test cases, a secure connection to `mysqld` is initiated for

those cases that require one. For those test cases that do not require SSL, an unencrypted connection is initiated.

- If `mysqld` does not support SSL, `mysql-test-run.pl` skips those test cases that require secure connections.

If `mysql-test-run.pl` is started with the `--ssl` option, it sets up a secure connection for all test cases. In this case, if `mysqld` does not support SSL, `mysql-test-run.pl` exits with an error message: `Couldn't find support for SSL`

2.3 How to Report Bugs from Tests in the MySQL Test Suite

If test cases from the test suite fail, you should do the following:

- Do not file a bug report before you have found out as much as possible about what when wrong. See the instructions at http://dev.mysql.com/doc/refman/en/bug-reports.
- Make sure to include the output of `mysql-test-run.pl`, as well as contents of all `.reject` files in the `mysql-test/r` directory, or the (often much shorter) diff that `mysql-test-run.pl` reported.
- Check whether an individual test in the test suite also fails when run on its own:

```
shell> cd mysql-test
shell> ./mysql-test-run.pl test_name
```

  If this fails, and you are compiling MySQL yourself, you should configure MySQL with `--with-debug` and run `mysql-test-run.pl` with the `--debug` option. If this also fails, open a bug report and upload the trace file `mysql-test/var/tmp/master.trace` to the report, so that we can examine it. For instructions, see How to Report Bugs or Problems. Please remember to also include a full description of your system, the version of the `mysqld` binary and how you compiled it.
- Run `mysql-test-run.pl` with the `--force` option to see whether any other tests fail.
- If you have compiled MySQL yourself, check the MySQL Reference Manual to see whether there are any platform-specific issues for your system. There might be configuration workarounds to deal with the problems that you observe. Also, consider using one of the binaries we have compiled for you at http://dev.mysql.com/downloads/. All our standard binaries should pass the test suite!
- If you get an error such as `Result length mismatch` or `Result content mismatch` it means that the output of the test was not an exact match for the expected output. This could be a bug in MySQL or it could be that your version of `mysqld` produces slightly different results under some circumstances.

  This output from `mysql-test-run.pl` should include a diff of the expected and actual result. If unable to produce a diff, it will instead print out both in full. In addition, the actual result is put in a file in the `r` directory with a `.result` extension.
- If a test fails completely, you should check the logs file in the `mysql-test/var/log` directory for hints of what went wrong.
- If you have compiled MySQL with debugging, you can try to debug test failures by running `mysql-test-run.pl` with either or both of the `--gdb` and `--debug` options.

  If you have not compiled MySQL for debugging you should probably do so by specifying the `-DWITH_DEBUG` option when you invoke `CMake`.

Chapter 3 Running Test Cases

Table of Contents

Typically, you run the test suite either from within a source tree (after MySQL has been built), or on a host where the MySQL server distribution has been installed. (If you find that the test suite is not included in your distribution, look for a similar distribution with `-test` in the name and install that as well.)

To run tests, your current working directory should be the `mysql-test` directory of your source tree or installed distribution. In a source distribution, `mysql-test` is under the root of the source tree. In a binary distribution, the location of `mysql-test` depends on the distribution layout. The program that runs the test suite, `mysql-test-run.pl`, will figure out whether you are in a source tree or an installed directory tree.

To run the test suite, change location into your `mysql-test` directory and invoke the `mysql-test-run.pl` script:

```
shell> cd mysql-test
shell> ./mysql-test-run.pl
```

`mysql-test-run.pl` accepts options on the command line. For example:

```
shell> ./mysql-test-run.pl --force --suite=binlog
```

By default, `mysql-test-run.pl` exits if a test case fails. `--force` causes execution to continue regardless of test case failure.

For a full list of the supported options, see Section 5.3, “`mysql-test-run.pl` — Run MySQL Test Suite”.

To run one or more specific test cases, name them on the `mysql-test-run.pl` command line. Test case files have names like `t/`*`test_name`*`.test`, where *`test_name`* is the name of the test case, but each name given on the command line should be the test case name, not the full test case file name. The following command runs the test case named `rpl_abcd`, which has a test file of `t/rpl_abcd.test`:

```
shell> ./mysql-test-run.pl rpl_abcd
```

To run a family of test cases for which the names share a common prefix, use the `--do-test` option:

```
shell> ./mysql-test-run.pl --do-test=prefix
```

For example, the following command runs the events tests (test cases that have names beginning with `events`):

```
shell> ./mysql-test-run.pl --do-test=events
```

To run a specific named testsuite with all the test cases in it, use the `--suite` option:

```
shell> ./mysql-test-run.pl --suite=suite_name
```

For example, the following command runs the replication tests located in the `rpl` suite:

```
shell> ./mysql-test-run.pl --suite=rpl
```

`mysql-test-run.pl` starts the MySQL server, sets up the environment for calling the `mysqltest` program, and invokes `mysqltest` to run the test case. For each test case to be run, `mysqltest` handles operations such as reading input from the test case file, creating server connections, and sending SQL statements to servers.

The language used in test case files is a mix of commands that the `mysqltest` program understands and SQL statements. Input that `mysqltest` doesn't understand is assumed to consist of SQL statements to be sent to the database server. This makes the test case language familiar to those that know how to write SQL and powerful enough to add the control needed to write test cases.

You need not start a MySQL server first before running tests. Instead, the `mysql-test-run.pl` program will start the server or servers as needed. Any servers started for the test run use ports in the range from 13000 by default.

3.1 Running Tests in Parallel

It is possible to run more than one instance of `mysql-test-run.pl` simultaneously on the same machine. Both will by default use server ports from 13000 but will coordinate used port numbers as well as check for availibility, to avoid conflicts.

Running several instances from the same `mysql-test` directory is possible but problematic. You must the use the `--vardir` to set different log directories for each instance. Even so, you can get into trouble becuse they will write `.reject` files to the same directories.

It is also possible to have a single `mysql-test-run.pl` run tests in several threads in parallel. Execution of the tests will be distributed among the threads. This is achieved using the `--parallel` option, with the number of threads as argument. The special value `auto` will ask `mysql-test-run.pl` to pick a value automatically, based on system information. The parallel option may also be given using the environment variable `MTR_PARALLEL`.

Chapter 4 Writing Test Cases

Table of Contents

Normally, you run the test suite during the development process to ensure that your changes do not cause existing test cases to break. You can also write new test cases or add tests to existing cases. This happens when you fix a bug (so that the bug cannot reappear later without being detected) or when you add new capabilities to the server or other MySQL programs.

This chapter provides guidelines for developing new test cases for the MySQL test framework.

Note

All test cases added to the MySQL source repository are published on the Internet. Take care that their contents include no confidential information, or copyrighted third-party material with a licence that would not allow this.

Some definitions:

- One “test file” is one “test case.”
- One “test case” might contain a “test sequence” (that is, a number of individual tests that are grouped together in the same test file).
- A “command” is an input test that `mysqltest` recognizes and executes itself. A “statement” is an SQL statement or query that `mysqltest` sends to the MySQL server to be executed.

4.1 Writing a Test Case: Quick Start

The basic principle of test case evaluation is that output resulting from running a test case is compared to the expected result. This is just a `diff` comparison between the output and an expected-result file that the test writer provides. This simplistic method of comparison does not by itself provide any way to handle variation in the output that may occur when a test is run at different times. However, the test language provides commands for postprocessing result output before the comparison occurs. This enables you to manage certain forms of expected variation.

Use the following procedure to write a new test case. In the examples, *`test_name`* represents the name of the test case. It is assumed here that you'll be using a development source tree, so that when you create a new test case, you can commit the files associated with it to the source repository for others to use.

1. Change location to the test directory `mysql-`*`version`*`/mysql-test`:

   ```
   shell> cd mysql-version/mysql-test
   ```

   `mysql-`*`version`* represents the root directory of your source tree, such as `mysql-5.6`.

2. Create the test case in a file `t/`*`test_name`*`.test`. You can do this with any text editor. For details of the language used for writing `mysqltest` test cases, see Chapter 6, *`mysqltest` Language Reference*.

3. Create an empty result file:

   ```
   shell> touch r/test_name.result
   ```

4. Run the test:

   ```
   shell> ./mysql-test-run.pl test_name
   ```

5. Assuming that the test case produces output, it should fail because the output does not match the result file (which is empty at this point). The failure results in creation of a reject file named `r/`*`test_name`*`.reject`. Examine this file. If the reject file appears to contain the output that you expect the test case to produce, copy its content to the result file:

   ```
   shell> cp r/test_name.reject r/test_name.result
   ```

   Another way to create the result file is by invoking `mysql-test-run.pl` with the `--record` option to record the test output in the result file:

   ```
   shell> ./mysql-test-run.pl --record test_name
   ```

6. Run the test again. This time it should succeed:

   ```
   shell> ./mysql-test-run.pl test_name
   ```

   You can also run the newly created test case as part of the entire suite:

   ```
   shell> ./mysql-test-run.pl
   ```

It is also possible to invoke the `mysqltest` program directly. If the test case file refers to environment variables, you will need to define those variables in your environment first. For more information about the `mysqltest` program, see Section 5.1, “`mysqltest` — Program to Run Test Cases”.

4.2 Test Case Coding Guidelines

4.2.1 File Naming and Organization Guidelines

Test case file names may use alphanumeric characters (`A-Z, a-z, 0-9`), underscore (`'_'`) or dash (`'-'`), but should not start with underscore or dash. Other special characters may work but this is not guaranteed so they should be avoided.

Test names have traditionally used lowercase only, and we recommend continuing this for tests added to the common repository, though uppercase letters are also supported.

Test cases are located in the `mysql-test/t` directory. Test case file names consist of the test name with a `.test` suffix. For example, a test named `foo` should be written in the file `mysql-test/t/foo.test`.

In addition to this directory, tests are organized in test suites, located in subdirectories under the `suite` directory. For example, a test named `bar` under the replication suite `rpl` may be stored in the file `mysql-test/suite/rpl/t/bar.test`.

In practice, the file would likely be called `rpl_bar.test` as tests in a suite usually also have the suite name as a prefix. This is just a convention from the time when suites were not supported, and not a requirement for test naming.

One test case file can be a collection of individual tests that belong together. If one of the tests fails, the entire test case fails. Although it may be tempting to write each small test into a single file, that will be too inefficient and makes test runs unbearably slow. So make the test case files not too big, not too small.

Each test case (that is, each test file) must be self contained and independent of other test cases. Do not create or populate a table in one test case and depend on the table in a later test case. If you have some common initialization that needs to be done for multiple test cases, create an include file. That is, create a file containing the initialization code in the `mysq-test/include` directory, and then put a `source` command in each test case that requires the code. For example, if several test cases need to have a given table created and filled with data, put the statements to do that in a file named `mysql-test/include/create_my_table.inc`. The `.inc` is a convention, not a requirement. Then put the following command in each test case file that needs the initialization code:

```
--source include/create_my_table.inc
```

The file name in the `source` command is relative to the `mysql-test` directory. Remember to drop the table at the end of each test that creates it.

4.2.2 Test Case Content-Formatting Guidelines

When you write a test case, please keep in mind the following general guidelines.

There are C/C++ coding guidelines in the MySQL Source Code documentation; please apply them when it makes sense to do so: Coding Guidelines

Other guidelines may be found in this page, which discusses general principles of test-case writing: MySQL Internals: How to Create Good Test Cases

The following guidelines are particularly applicable to writing test cases:

- To write a test case file, use any text editor that uses linefeed (newline) as the end-of-line character.
- Avoid lines longer than 80 characters unless there is no other choice.
- A comment in a test case is started with the "#" character.

  Section 6.1, "`mysqltest` Input Conventions", discusses the details of input syntax for `mysqltest` test cases.
- Use spaces, not tabs.
- Write SQL statements using the same style as our manual:
  - Use uppercase for SQL keywords.
  - Use lowercase for identifiers (names of objects such as databases, tables, columns, and so forth).

  Ignore this guidline if your intent is to test lettercase processing for SQL statements, of course.

  Use lowercase for `mysqltest` commands (`echo`, `sleep`, `let`, and so forth).

  You may notice that many existing test cases currently do not follow the lettercase guideline and contain SQL statements written entirely in lowercase. Nevertheless, *please use the guideline for new tests*. Lettercase for older tests can be left as is, unless perhaps you need to revise them significantly.
- Break a long SQL statement into multiple lines to make it more readable. This means that you will need to write it using a "`;`" delimiter at the end of the statement rather than using "`--`" at the beginning because the latter style works only for single-line statements.
- Include comments. They save the time of others. In particular:
  - Please include a header in test files that indicates the purpose of the test and references the corresponding worklog task, if any.
  - Comments for a test that is related to a bug report should include the bug number and title.

  Worklog and bug numbers are useful because they enable people who are interested in additional background related to the test case to know which worklog entries or bug reports to read.

Example SQL statement, formatted onto multiple lines for readability:

```
SELECT f1 AS "my_column", f10 ....
FROM mysqltest1.t5
WHERE (f2 BETWEEN 17 AND 25 OR f2 = 61)
  AND f3 IN (SELECT ....
             FROM mysqltest1.t4
             WHERE .....)
ORDER BY ... ;
```

Example test file header:

```
############ suite/funcs_1/t/a_processlist_val_no_prot.test #############
#                                                                        #
# Testing of values within INFORMATION_SCHEMA.PROCESSLIST                #
#                                                                        #
# The prepared statement variant of this test is                         #
# suite/funcs_1/t/b_processlist_val_ps.test.                             #
```

```
#                                                                           #
# There is important documentation within                                   #
#       suite/funcs_1/datadict/processlist_val.inc                          #
#                                                                           #
# Note(mleich):                                                             #
#    The name "a_process..." with the unusual prefix "a_" is                #
#    caused by the fact that this test should run as second test, that      #
#    means direct after server startup and a_processlist_priv_no_prot.      #
#    Otherwise the connection IDs within the processlist would differ.      #
#                                                                           #
# Creation:                                                                 #
# 2007-08-09 mleich Implement this test as part of                          #
#                   WL#3982 Test information_schema.processlist             #
#                                                                           #
#############################################################################
```

Example test reference to bug report:

```
# Bug #3671 Stored procedure crash if function has "set @variable=param"
```

4.2.3 Naming Conventions for Database Objects

It is possible to run test cases against a production server. This is very unlikely to happen by accident, as `mysql-test-run.pl` will start its own server unless you use the `--extern` Even so, try to write test cases in a way that reduces the risk that running tests will alter or destroy important tables, views, or other objects. (`DROP DATABASE` statements are particularly dangerous if written using names that could exist on a customer's machine.) To avoid such problems, we recommend the following naming conventions:

- User names: User names should begin with “mysql” (for example, `mysqluser1`, `mysqluser2`)
- Database names: Unless you have a special reason not to, use the default database named `test` that is already created for you. For tests that need to operate outside the `test` database, database names should contain “test” or begin with “mysql” (for example, `mysqltest1`, `mysqltest2`)
- Table names: `t1`, `t2`, `t3`, ...
- View names: `v1`, `v2`, `v3`, ...

For examples of how to name objects, examine the existing test cases. Of course, you can name columns and other objects inside tables as you wish.

4.3 Sample Test Case

Here is a small sample test case:

```
--disable_warnings
DROP TABLE IF EXISTS t1;
--enable_warnings
SET SQL_WARNINGS=1;

CREATE TABLE t1 (a INT);
INSERT INTO t1 VALUES (1);
INSERT INTO t1 VALUES ("hej");
```

The first few lines try to clean up from possible earlier runs of the test case by dropping the `t1` table. The test case uses `disable_warnings` to prevent warnings from being written to the output because it is not of any interest at this point during the test to know whether the table `t1` was there. After dropping the table, the test case uses `enable_warnings` so that subsequent warnings will be written to the output. The test case also enables verbose warnings in MySQL using the `SET SQL_WARNINGS=1;` statement.

Next, the test case creates the table `t1` and tries some operations. Creating the table and inserting the first row are operations that should not generate any warnings. The second insert should generate a warning because it inserts a nonnumeric string into a numeric column. The output that results from running the test looks like this:

```
DROP TABLE IF EXISTS t1;
SET SQL_WARNINGS=1;
CREATE TABLE t1 (a INT);
INSERT INTO t1 VALUES (1);
INSERT INTO t1 VALUES ("hej");
Warnings:
Warning 1265    Data truncated for column 'a' at row 1
```

Note that the result includes not only the output from SQL statements, but the statements themselves. Statement logging can be disabled with the `disable_query_log` test language command. There are several options for controlling the amount of output from running the tests.

If there was a test failure, it will be reported to the screen. You can see the actual output from the last unsuccessful run of the test case in the reject file `r/test_name.reject`.

4.4 Cleaning Up from a Previous Test Run

For efficiency, the `mysqltest` test engine does not start with a clean new database for running each test case, so a test case generally starts with a “cleaning up section.” Assume that a test case will use two tables named `t1` and `t2`. The test case should begin by making sure that any old tables with those names do not exist:

```
--disable_warnings
DROP TABLE IF EXISTS t1,t2;
--enable_warnings
```

The `disable_warnings` command instructs the test engine not to log any warnings until an `enable_warnings` command occurs or the test case is ended. (MySQL generates a warning if the table `t1` or `t2` does not exist.) Surrounding this part of the test case with commands to disable and enable warnings makes its output the same regardless of whether the tables exist before the test is started. After ensuring that the tables do not exist, we are free to put in any SQL statements that create and use the tables `t1` and `t2`. The test case should also clean up at the end of the test by dropping any tables that it creates.

Let's put in some SQL code into this test case:

```
--disable_warnings
DROP TABLE IF EXISTS t1,t2;
--enable_warnings
CREATE TABLE t1 (
  Period SMALLINT(4) UNSIGNED ZEROFILL DEFAULT '0000' NOT NULL,
  Varor_period SMALLINT(4) UNSIGNED DEFAULT '0' NOT NULL
);
CREATE TABLE t2 (Period SMALLINT);

INSERT INTO t1 VALUES (9410,9412);
INSERT INTO t2 VALUES (9410),(9411),(9412),(9413);

SELECT PERIOD FROM t1;
SELECT * FROM t1;
SELECT t1.* FROM t1;
SELECT * FROM t1 INNER JOIN t2 USING (Period);
```

```
DROP TABLE t1, t2;
```

If a test case creates other objects such as stored programs or user accounts, it should take care to also clean those up at the beginning and end of the test. Temporary files should also be removed, either at the end or just after they have been used.

4.5 Generating a Test Case Result File

The test code we just wrote contains no checks of the result. The test will report a failure for one of two reasons:

- An individual SQL statement fails with an error
- The overall test case result does not match what was expected

Note that these are the reasons why `mysqltest` would fail; if the test is run from `mysql-test-run.pl` the test may fail for additional reasons.

In the first case, `mysqltest` aborts with an error. The second case requires that we have a record of the expected result so that it can be compared with the actual result. To generate a file that contains the test result, run the test with the `--record` option, like this:

```
shell> cd mysql-test
shell> ./mysql-test-run.pl --record foo
```

Running the test as shown creates a result file named `mysql-test/r/foo.result` that has this content:

```
DROP TABLE IF EXISTS t1,t2;
CREATE TABLE t1 (
Period SMALLINT(4) UNSIGNED ZEROFILL DEFAULT '0000' NOT NULL,
Varor_period SMALLINT(4) UNSIGNED DEFAULT '0' NOT NULL
);
CREATE TABLE t2 (Period SMALLINT);
INSERT INTO t1 VALUES (9410,9412);
INSERT INTO t2 VALUES (9410),(9411),(9412),(9413);
SELECT period FROM t1;
period
9410
SELECT * FROM t1;
Period  Varor_period
9410    9412
SELECT t1.* FROM t1;
Period  Varor_period
9410    9412
SELECT * FROM t1 INNER JOIN t2 USING (Period);
Period  Varor_period
9410    9412
DROP TABLE t1, t2;
ok
```

If we look at this result file, it contains the statements in the `foo.test` file together with the output from the `SELECT` statements. The output for each statement includes a row of column headings followed by data rows. Rows have columns separated by Tab characters.

At this point, you should inspect the result file and determine whether its contents are as expected. If so, let it be part of your test case. If the result is not as expected, you have found a problem, either with the server or the test. Determine the cause of the problem and fix it. For example, the test might produce output that varies from run to run. To deal with this, you can postprocess the output before the comparison occurs. See Section 4.8, “Dealing with Output That Varies Per Test Run”.

4.6 Checking for Expected Errors

A good test suite checks not only that operations succeed as they ought, but also that they fail as they ought. For example, if a statement is illegal, the server should reject it with an error message. The test suite should verify that the statement fails and that it fails with the proper error message.

The test engine enables you to specify “expected failures.” Let's say that after we create `t1`, we try to create it again without dropping it first:

```
--disable_warnings
DROP TABLE IF EXISTS t1,t2;
--enable_warnings
CREATE TABLE t1 (
  Period SMALLINT(4) UNSIGNED ZEROFILL DEFAULT '0000' NOT NULL,
  Varor_period SMALLINT(4) UNSIGNED DEFAULT '0' NOT NULL
);
CREATE TABLE t2 (Period SMALLINT);

INSERT INTO t1 VALUES (9410,9412);
INSERT INTO t2 VALUES (9410),(9411),(9412),(9413);

SELECT period FROM t1;
SELECT * FROM t1;
SELECT t1.* FROM t1;
SELECT * FROM t1 INNER JOIN t2 USING (Period);

CREATE TABLE t1 (something SMALLINT(4));
```

The result is failure and an error:

```
At line 21: query 'CREATE TABLE t1 (something SMALLINT(4))'
failed: 1050: Table 't1' already exists
```

To handle this error and indicate that indeed we do expect it to occur, we can put an `error` command before the second `create table` statement. Either of the following commands test for this particular MySQL error:

```
--error 1050
--error ER_TABLE_EXISTS_ERROR
```

1050 is the numeric error code and `ER_TABLE_EXISTS_ERROR` is the symbolic name. Symbolic names are more stable than error numbers because the numbers sometimes change, particularly for those created during recent development. For such errors, use of numbers rather than the names in a test case will require test to be revised should the numbers change.

After we make a change to add an `error` command before the `CREATE TABLE` statement and run the test again, the end of the result will look like this:

```
CREATE TABLE t1 (something SMALLINT(4));
ERROR 42S01: Table 't1' already exists
```

In this case, the result shows the statement that causes the error, together with the resulting error message. The fact that `mysqltest` does not terminate and that the error message becomes part of the result indicates that the error was expected.

You can also test for errors by specifying an SQLSTATE value. For MySQL error number 1050, the corresponding SQLSTATE value is 42S01. To specify an SQLSTATE value in an `error` command, use an `S` prefix:

```
--error S42S01
```

A disadvantage of SQLSTATE values is that sometimes they correspond to more than one MySQL error code. Using the SQLSTATE value in this case might not be specific enough (it could let through an error that you do not actually expect).

If you want to test for multiple errors, the `error` command allows multiple arguments, separated by commas. For example:

```
--error ER_NO_SUCH_TABLE,ER_KEY_NOT_FOUND
```

For a list of MySQL error codes, symbolic names, and SQLSTATE values, see http://dev.mysql.com/doc/refman/en/error-messages-server.html. You can also examine the `mysqld_error.h` and `sql_state.h` files in the `include` directory of a MySQL source distribution.

As of MySQL 8.0, it is also possible to use symbolic error names to refer to client errors:

```
--error CR_SERVER_GONE_ERROR
```

For a list of MySQL client error codes, see http://dev.mysql.com/doc/refman/en/error-messages-client.html. You can also examine the `errmsg.h` file in the `include` directory of a MySQL source distribution.

The built-in variable `$mysql_errno` contains the numeric error returned by the most recent SQL statement sent to the server, or 0 if the statement executed successfully. This may be useful after statements that may or may not fail, or fail in more than one way (more than one argument to the `error` command), in case you need to perform different actions. Note that this applies to SQL statements, not to other commands.

From MySQL 5.5.17, there is also a variable `$mysql_errname` which contains the symbolic name of the last error. In some cases the symbolic name is not available; in those cases the variable will contain the string `"<Unknown>"`. For new test development we recommend testing against the name rather than the number, since the number may change in future versions. If the last statement succeeded (`$mysql_errno` is 0), this variable is an empty string.

4.7 Controlling the Information Produced by a Test Case

By default, the `mysqltest` test engine produces output only from `select`, `show`, and other SQL statements that you expect to produce output (that is, statements that create a result set). It also produces output from certain commands such as `echo` and `exec`. `mysqltest` can be instructed to be more or less verbose.

Suppose that we want to include in the result the number of rows affected by or returned by SQL statements. To do this, add the following line to the test case file preceding the first table-creation statement:

```
--enable_info
```

After rerunning the test by invoking `mysql-test-run.pl` with the `--record` option to record the new result, the result file will contain more information:

```
DROP TABLE IF EXISTS t1,t2;
CREATE TABLE t1 (
```

```
Period SMALLINT(4) UNSIGNED ZEROFILL DEFAULT '0000' NOT NULL,
Varor_period SMALLINT(4) UNSIGNED DEFAULT '0' NOT NULL
);
affected rows: 0
CREATE TABLE t2 (Period SMALLINT);
affected rows: 0
INSERT INTO t1 VALUES (9410,9412);
affected rows: 1
INSERT INTO t2 VALUES (9410),(9411),(9412),(9413);
affected rows: 4
info: Records: 4  Duplicates: 0  Warnings: 0
SELECT period FROM t1;
period
9410
affected rows: 1
SELECT * FROM t1;
Period  Varor_period
9410    9412
affected rows: 1
SELECT t1.* FROM t1;
Period  Varor_period
9410    9412
affected rows: 1
SELECT * FROM t1 INNER JOIN t2 USING (Period);
Period  Varor_period
9410    9412
affected rows: 1
DROP TABLE t1, t2;
affected rows: 0
ok
```

To turn off the affected-rows reporting, add this command to the test case file:

```
--disable_info
```

In general, options can be enabled and disabled for different parts of the test file. Suppose that we are interested in the internals of the database as well. We could enable the display of query metadata using `enable_metadata`. With this option enabled, the test output is a bit verbose. However, as mentioned earlier, the option can be enabled and disabled selectively so that it is enabled only for those parts of the test case where it interests you to know more.

If you perform an operation for which you have no interest in seeing the statements logged to the result, you can disable statement logging. For example, you might be initializing a table where you don't really expect a failure, and you are not interested in seeing the initialization statements in the test result. You can use the `disable_query_log` command to temporarily disable recording of input SQL statements, and enable recording again with `enable_query_log`. You can disable the recording of the output from executing commands using `disable_result_log` and enable recording again with `enable_result_log`.

4.8 Dealing with Output That Varies Per Test Run

It is best to write each test case so that the result it produces does not vary for each test run, or according to factors such as the time of day, differences in how program binaries are compiled, the operating system, and so forth. For example, if the result contains the current date and time, the test engine has no way to verify that the result is correct.

However, sometimes a test result is inherently variable according to external factors, or perhaps there is a part of a result that you simply do not care about. `mysqltest` provides commands that enable you to postprocess test output into a more standard format so that output variation across test runs will not trigger a result mismatch.

One such command is `replace_column`, which specifies that you want to replace whatever is in a given column with a string. This makes the output for that column the same for each test run.

To see how this command works, add the following row after the first insert in the test case:

```
INSERT INTO t1 VALUES (DATE_FORMAT(NOW(), '%s'),9999);
```

Then record the test result and run the test again:

```
shell> ./mysql-test-run.pl --record foo
shell> ./mysql-test-run.pl foo
```

Most likely, a failure will occur and `mysql-test-run.pl` will display the difference between the expected result and what we actually got, like this (the header has been simplified):

```
CURRENT_TEST: main.foo
--- r/foo.result        2009-11-17 16:22:38
+++ r/foo.reject        2009-11-17 16:22:47
@@ -10,15 +10,15 @@
 SELECT period FROM t1;
 period
 9410
-0038
+0047
 SELECT * FROM t1;
 Period Varor_period
 9410   9412
-0038   9999
+0047   9999
 SELECT t1.* FROM t1;
 Period Varor_period
 9410   9412
-0038   9999
+0047   9999
 SELECT * FROM t1 INNER JOIN t2 USING (Period);
 Period Varor_period
 9410   9412

mysqltest: Result content mismatch
```

The actual numbers will likely be different for your case, and the format of the diff may also vary.

If we are not really interested in the first column, one way to eliminate this mismatch is by using the `replace_column` command. The duration of the effect of this command is the next SQL statement, so we need one before each `select` statement:

```
--replace_column 1 SECONDS
SELECT period FROM t1;
--replace_column 1 SECONDS
SELECT * FROM t1;
--replace_column 1 SECONDS
SELECT t1.* FROM t1;
```

In the `replace_column` commands, `SECONDS` could be any string. Its only purpose is to map variable output onto a constant value. If we record the test result again, we will succeed each time we run the test after that. The result file will look like this:

```
DROP TABLE IF EXISTS t1,t2;
CREATE TABLE t1 (
```

```
Period SMALLINT(4) UNSIGNED ZEROFILL DEFAULT '0000' NOT NULL,
Varor_period SMALLINT(4) UNSIGNED DEFAULT '0' NOT NULL
);
affected rows: 0
CREATE TABLE t2 (Period SMALLINT);
affected rows: 0
INSERT INTO t1 VALUES (9410,9412);
affected rows: 1
INSERT INTO t1 VALUES (DATE_FORMAT(NOW(), '%s'),9999);
affected rows: 1
INSERT INTO t2 VALUES (9410),(9411),(9412),(9413);
affected rows: 4
info: Records: 4  Duplicates: 0  Warnings: 0
SELECT period FROM t1;
period
SECONDS
SECONDS
affected rows: 2
SELECT * FROM t1;
Period  Varor_period
SECONDS 9412
SECONDS 9999
affected rows: 2
SELECT t1.* FROM t1;
Period  Varor_period
SECONDS 9412
SECONDS 9999
affected rows: 2
SELECT * FROM t1 INNER JOIN t2 USING (Period);
Period  Varor_period
9410    9412
affected rows: 1
DROP TABLE t1, t2;
affected rows: 0
ok
```

4.9 Passing Options from mysql-test-run.pl to mysqld or mysqltest

mysql-test-run.pl supports several options that enable you to pass options to other programs. Each of these options takes a value consisting of one or more comma-separated options:

- The --mysqld option tells mysql-test-run.pl to start the mysqld server with the named option added. More than one such extra option may be provided. The following command causes --skip-innodb and --key_buffer_size=16384 to be passed to mysqld:

  ```
  shell> mysql-test-run.pl --mysqld=--skip-innodb --mysqld=--key_buffer_size=16384
  ```

  Note how --mysqld needs to be repeated for each server option to add. It does not work to add several server options with one --mysqld even if enclosed in quotation marks, as that will be interpreted as a single server option (including spaces).

- The --combination option is similar to --mysqld, but behaves differently. mysql-test-run.pl executes multiple test runs, using the options for each instance of --combination in successive runs. The following command passes --skip-innodb to mysqld for the first test run, and --innodb and --innodb-file-per-table for the second test run:

  ```
  shell> mysql-test-run.pl
           --combination=--skip-innodb
           --combination=--innodb,--innodb-file-per-table
  ```

If `--combination` is given only once, it has no effect.

For test runs specific to a given test suite, an alternative to the use of the `--combination` option is to create a `combinations` file in the suite directory. The file should contain a section of options for each test run. For an example, see Section 4.13.1, “Controlling the Binary Log Format Used for an Entire Test Run”.

- The `--mysqltest` option is used to pass extra options to `mysqltest`.

```
shell> mysql-test-run.pl --mysqltest=options
```

For an example, see Section 5.3, “`mysql-test-run.pl` — Run MySQL Test Suite”.

4.10 Specifying Test Case-Specific Server Options

Within a test case, many system variables can be set by using statements such as these:

```
SET sql_warnings=1;
SET sql_mode='NO_AUTO_VALUE_ON_ZERO';
```

But sometimes you need to restart the server to use command-line options that are specific to a given test case. You can specify these options in a file named `mysql-test/t/`*`test_name`*`-master.opt`. When a file named `t/`*`test_name`*`-master.opt` exists, `mysql-test-run.pl` examines it for extra options that the server needs to be run with when executing the *`test_name`* test case. If no server has yet been started or the current server is running with different options, `mysql-test-run.pl` restarts the server with the new options.

You may refer to environment variables in the option file, using the usual *`$VAR_NAME`* syntax. From MySQL 5.6, it is also possible to refer to optional variables using the syntax *`$?VAR_NAME`*. This will be replaced with an empty string if the variable is not set.

As a special case, the option `--skip-core-file` will be interpreted by `mysql-test-run.pl`, which will then block the server from producing any core files or crash dumps during this test. This may be useful for tests that intentionally crash the server.

Another special case is `--testcase-timeout=`*`minutes`* which can be used to set a different, longer timeout for a particular test case. The given timeout in minutes will be used for this test if it's longer than the default.

Files in the `mysql-test/t` directory with names ending in `-slave.opt` are similar, but they are used for slave servers in replication tests.

Sometimes it's also necessary to execute some external commands before starting the server, such as creating a directory. If you add a file named `t/`*`test_name`*`-master.sh`, it will be executed by `mysql-test-run.pl` before it starts the server; a similar file may be created for the slave in replication tests.

Because the `.sh` file is executed through `/bin/sh`, it cannot be used on Windows, and any tests using such a file will automatically be skipped if you run on Windows. For this reason, this mechanism may be replaced with a more portable one in some future release of MySQL.

4.11 Specifying Test Case-Specific Bootstrap Options

If a test has to run with a particular value of a bootstrap variable such as `--innodb-page-size` or `--innodb-data-file-path`, the option can be passed on the command line while running `mysql-test-`

`run.pl`. For example, consider a test that can only run with `--innodb-page-size=8k`. The test can be run like this:

```
shell>  perl mysql-test-run.pl test_name_8k --bootstrap=--innodb-page-size=8k --mysqld=--innodb-page-size=8k
```

To ensure the test runs with only a particular page size, an inc file is used to make it skip during runs with other page sizes.

```
--source include/have_innodb_8k.inc
```

One limitation of using this include file is that the test is skipped unless the required option is passed on the command line.

Another way to run a test with a bootstrap variable is to delete the existing data directory, and initialize a new data directory with the bootstrap options, inside the test. When the server is started with the bootstrap options after this, the SQL queries will run with the specified options.

Now it is possible to pass bootstrap options in the master.opt file of the test, so that the test can run with the specified value of the bootstrap options without using any command line arguments, or reinitializing the server within the test. Specifying bootstrap variables in the opt file is the preferred method. The usage is:

```
--bootstrap --innodb-page-size=8k
```

or

```
--bootstrap=--innodb-page-size=8k
```

Each bootstrap variable must be specified as the value of a `--bootstrap` option in the opt file to ensure `mysql-test-run.pl` recognizes that the variable must be used during server initialization. If there are bootstrap options in the file, the current data directory is deleted, and initialized again with the options set in the file. The server is also started with the bootstrap options passed along with the other options in the opt file.

Similarly, bootstrap options passed in the slave.opt will be used to reinitialize the slave server in replication scenarios.

Note that since the options are passed in the opt files, they have precedence over the command line options. So, if a test has a master.opt file containing `--innodb-page-size=8k`, and while it is being run, `--innodb-page-size=4k` is passed on the command line, the test will run with 8k page size.

4.12 Using Include Files to Simplify Test Cases

The `include` directory contains many files intended for inclusion into test case files. For example, if a test case needs to verify that the server supports the `CSV` storage engine, use this line in the test case file:

```
--source include/have_csv.inc
```

These include files serve many purposes, but in general, they encapsulate operations of varying complexity into single files so that you can perform each operation in a single step. Include files are available for operations such as these:

- Ensure that a given feature is available. The file checks to make sure that the feature is available and exits if not.

  - Storage engine tests: These files have names of the form `have_engine_name.inc`, such as `have_innodb.inc` or `have_falcon.inc`. The `MyISAM`, `MERGE`, and `MEMORY` storage engines are always supported and need not be checked.
  - Character set tests: These files have names of the form `have_charset_name.inc`, such as `have_utf8.inc` or `have_cp1251.inc`.
  - Debugging capabilities: Include the `have_debug.inc` file if a test requires that the server was built for debugging (that is, that the MySQL distribution was configured with the `--with-debug` option).
- Wait for a condition to become true. Set the `$wait_condition` variable to an SQL statement that selects a value and then include the `wait_condition.inc` file. The include file executes the statement in a loop with a 0.1 second sleep between executions until the select value is nonzero. For example:

```
let $wait_condition= SELECT c = 3 FROM t;
--source include/wait_condition.inc
```

- Control the binary log format. See Section 4.13, “Controlling the Binary Log Format Used for Tests”.
- Control replication slave servers. See Section 4.14, “Writing Replication Tests”.

You can think of an include file as a rudimentary form of subroutine that is “called” at the point of inclusion. You can “pass parameters” by setting variables before including the file and referring to them within the file. You can “return” values by setting variables within the file and referring them following inclusion of the file.

4.13 Controlling the Binary Log Format Used for Tests

The server can do binary logging using statement-based logging (SBL), which logs events as statements that produce data changes, or row-based logging (RBL), which logs events as changes to individual rows. It also supports mixed logging, which switches between SBL and RBL automatically as necessary.

The server's global `binlog_format` system variable indicates which log format is in effect. It has possible values of `STATEMENT`, `ROW`, and `MIXED` (not case sensitive). This system variable can be set at server startup by specifying `--binlog_format=value` on the command line or in an option file. A user who has the `SUPER` privilege can change the log format at runtime. For example:

```
SET GLOBAL binlog_format = STATEMENT;
```

Some tests require a particular binary log format. You can exercise control over the binary log format in two ways:

- To control the log format that the server uses for an entire test run, you can pass options to `mysql-test-run.pl` that tell it which format `mysqld` should use.
- To verify that a particular log format is in effect for a specific test case, you can use an appropriate include file that checks the current format and exits if the format is other than what is required.

The following sections describe how to use these techniques.

4.13.1 Controlling the Binary Log Format Used for an Entire Test Run

To specify the binary log format for a test run, you can use the `--mysqld` or `--combination` option to tell `mysql-test-run.pl` to pass a logging option to `mysqld`. For example, the following command runs

the tests from the `rpl` suite that have names that begin with `rpl_row`. The tests are run once with the binary log format set to `STATEMENT`:

```
shell> mysql-test-run.pl --suite=rpl --do-test=rpl_row
         --mysqld=--binlog_format=statement
```

To run tests under multiple log formats, use two or more instances of the `--combination` option. The following command runs the same tests as the preceding command, but runs them once with the binary log format set to `ROW` and a second time with the format set to `MIXED`:

```
shell> mysql-test-run.pl --suite=rpl --do-test=rpl_row
         --combination=--binlog_format=row
         --combination=--binlog_format=mixed
```

The `--combination` option must be given at least twice or it has no effect.

As an alternative to using the `--combination` option, you can create a file named `combinations` in the test suite directory and list the options that you would specify using `--combination`, one line per option. For the preceding `mysql-test-run.pl` command, the suite name is `rpl`, so you would create a file named `suite/rpl/combinations` with these contents:

```
[row]
--binlog_format=row

[mixed]
--binlog_format=mixed
```

Then invoke `mysql-test-run.pl` like this:

```
shell> mysql-test-run.pl --suite=rpl --do-test=row
```

The format of the `combinations` file is similar to that of `my.cnf` files (section names followed by options for each section), but options listed in the `combinations` file should include the leading dashes. (Options in `my.cnf` files are given without the leading dashes.) `mysql-test-run.pl` displays the section name following the test name when it reports the test result.

Any `--combination` options specified on the command line override those found in a `combinations` file.

The `--combination` option and the `combinations` file have different scope. The `--combination` option applies globally to all tests run by a given invocation of `mysql-test-run.pl`. The `combinations` file is placed in a test suite directory and applies only to tests in that suite.

4.13.2 Specifying the Required Binary Log Format for Individual Test Cases

To specify within a test case that a particular binary log format is required, include one of the following lines to indicate the format:

```
--source include/have_binlog_format_row.inc
--source include/have_binlog_format_statement.inc
--source include/have_binlog_format_mixed.inc
```

The following files can be used for tests that support two binary log formats:

```
--source include/have_binlog_format_mixed_or_row.inc
```

```
--source include/have_binlog_format_mixed_or_statement.inc
--source include/have_binlog_format_row_or_statement.inc
```

Before `mysql-test-run.pl` runs the test case, it checks whether the value that it is using for the `binlog_format` system variable matches what the test requires, based on whether the test refers to one of the preceding include files. If `binlog_format` does not have an appropriate value, `mysql-test-run.pl` skips the test.

If a test supports all binary log formats, none of the `have_binlog_format_*.inc` include files should be used in the test file. A test that includes no such file is assumed to support all formats.

4.14 Writing Replication Tests

If you are writing a replication test case, the test case file should begin with this command:

```
--source include/master-slave.inc
```

To switch between the master and slave, use these commands:

```
connection master;
connection slave;
```

If you need to do something on an alternative connection, you can use `connection master1;` for the master and `connection slave1;` for the slave.

To run the master with additional options for your test case, put them in command-line format in `t/test_name-master.opt`. When a file named `t/test_name-master.opt` exists, `mysql-test-run.pl` examines it for extra options that the server needs to be run with when executing the *`test_name`* test case. If no server has yet been started or the current server is running with different options, `mysql-test-run.pl` restarts the server with the new options.

For the slave, similar principles apply, but you should list additional options in `t/test_name-slave.opt`.

Several include files are available for use in tests that enable better control over the behavior of slave server I/O and SQL threads. The files are located in the `include` directory and have names of the form `wait_for_slave_*.inc`. By using these files, you can help make replication tests more stable because it will be more likely that test failures are due to replication failures, not due to problems with the tests themselves.

The slave-control include files address the issue that it is not always sufficient to use a `START SLAVE` or `STOP SLAVE` statement by itself: When the statement returns, the slave may not have reached the desired operational state. For example, with `START SLAVE`, the following considerations apply:

- It is not necessary to wait for the SQL thread after `START SLAVE` or `START SLAVE SQL_THREAD` because the thread will have started by the time statement returns.
- By contrast, it is necessary to wait for the I/O thread after `START SLAVE` or `START SLAVE IO_THREAD` because although the thread will have started when the statement returns, it may not yet have established the connection to the master.

To verify that a slave has reached the desired state, combine the use of `START SLAVE` or `STOP SLAVE` with an appropriate “wait” include file. The file contains code that waits until the state has been reached or a timeout occurs. For example, to verify that both slave threads have started, do this:

```
START SLAVE;
--source include/wait_for_slave_to_start.inc
```

Similarly, to stop both slave threads, do this:

```
STOP SLAVE;
--source include/wait_for_slave_to_stop.inc
```

The following list describes the include files that are available for slave control:

- `wait_for_slave_to_start.inc`

  Waits for both slave threads (I/O and SQL) to start. Should be preceded by a `START SLAVE` statement.

- `wait_for_slave_to_stop.inc`

  Waits for both slave threads (I/O and SQL) to stop. Should be preceded by a `STOP SLAVE` statement.

- `wait_for_slave_sql_to_stop.inc`

  Waits for the slave SQL thread to stop. Should be preceded by a `STOP SLAVE SQL_THREAD` statement.

- `wait_for_slave_io_to_stop.inc`

  Waits for the slave I/O thread to stop. Should be preceded by a `STOP SLAVE IO_THREAD` statement.

- `wait_for_slave_param.inc`

  Waits until `SHOW SLAVE STATUS` output contains a given value or a timeout occurs. Before including the file, you should set the `$slave_param` variable to the column name to look for in `SHOW SLAVE STATUS` output, and `$slave_param_value` to the value that you are waiting for the column to have.

  Example:

  ```
  let $slave_param= Slave_SQL_Running;
  let $slave_param_value= No;
  --source include/slave_wait_slave_param.inc
  ```

- `wait_for_slave_sql_error.inc`

  Waits until the SQL thread for the current connection has gotten an error or a timeout occurs. Occurrence of an error is determined by waiting for the `Last_SQL_Errno` column of `SHOW SLAVE STATUS` output to have a nonzero value.

4.15 Thread Synchronization in Test Cases

The Debug Sync facility allows placement of synchronization points in the code. They can be activated by statements that set the `debug_sync` system variable. An active synchronization point can emit a signal or wait for a signal to be emitted by another thread. This waiting times out after 300 seconds by default. The `--debug-sync-timeout=N` option for `mysql-test-run.pl` changes that timeout to *`N`* seconds. A timeout of zero disables the facility altogether, so that synchronization points will not emit or wait for signals, even if activated.

The purpose of the timeout is to avoid a complete lockup in test cases. If for some reason the expected signal is not emitted by any thread, the execution of the affected statement will not block forever. A warning shows up when the timeout happens. That makes a difference in the test result so that it will not go undetected.

For test cases that require the Debug Sync facility, include the following line in the test case file:

```
--source include/have_debug_sync.inc
```

For a description of the Debug Sync facility and how to use synchronization points, see MySQL Internals: Test Synchronization.

4.16 Suppressing Errors and Warning

After a test is finished, and if it didn't fail for some other reason, `mysql-test-run.pl` will check the log written by the server(s) during the test for any warnings or errors. If it finds any, it will print them out and the test will be reported as failed.

However, many warnings and also a few errors may occur normally without signifying a real problem. Also, many tests purposefully perform actions that will provoke warnings. For these not to result in a test failure, it is possible to specify that certain messages are to be suppressed.

There is a list of global suppressions; warnings that will always be suppressed. These are listed in the file `include/mtr_warnings.sql`. Any error or warning that contains one of the listed strings will be ignored. It is not necessary to include the whole text, and regular expressions can be used, such as `.*` to match any substring or `[0-9]*` to match any number.

You will rarely need to change or add to this global list, but you may need to suppress a particular error or warning for a new test case. A typical case is a test that performs some illegal action on purpose to test the respone; if this also results in a warning in the server log, this should not cause this particular test to fail.

To add a suppresion for a warning text like for example `The table 't23456' is full` where the number in the table name can vary, add this line anywhere in the test case file:

```
call mtr.add_suppression("The table 't[0-9]*' is full");
```

This may be put anythere in the test, but we recommend putting it either near the top, or just after the action that may result in the warning. If you're adding this line to an existing test, keep in mind that it will be echoed in the output, so the exact same line also needs to be added to the corresponding place in the result file. Alternatively, you may turn off logging like this, and will then not have to edit the result file:

```
--disable_query_log
call mtr.add_suppression("The table 't[0-9]*' is full");
--enable_query_log
```

It is also possible to intruct `mysql-test-run.pl` to skip the check for errors and warnings completely, by use of the `--nowarnings` command line option.

4.17 Stopping a Server During a Test

If a server dies during execution of a test case, this will be interpreted as a failure. However, there may be cases where you actually want to stop and possibly restart a server intentionally. It is possible to let the system know you expect a server to terminate, and to either wait or have it restarted immediately:

Before you initiate the action that will stop the server, the test case should write either `restart` or `wait` to the file `$MYSQLTEST_VARDIR/tmp/mysqld.1.expect`. The easiest way to do this is using the `exec echo` construct like in the following example.

If you write `restart` into this file, the server will be immediately restarted. If you write `wait` instead, the server will remain down, but can be restarted at a later time by writing `restart` to the file.

```
```

```
--exec echo "wait" > $MYSQLTEST_VARDIR/tmp/mysqld.1.expect
--shutdown_server 10
--source include/wait_until_disconnected.inc
# Do something while server is down
--enable_reconnect
--exec echo "restart" > $MYSQLTEST_VARDIR/tmp/mysqld.1.expect
--source include/wait_until_connected_again.inc
```

For your convenience, there is a file `include/restart_mysqld.inc` which you can "source" in your own test case to do what's shown in this code example.

The file name to write the command to will be `mysqld.2.expect` for the slave server in replication tests. Note that you have to use `$MYSQLTEST_VARDIR/tmp` for the directory here; if you use `$MYSQL_TMP_DIR` instead, it will not work when running tests in parallel.

It is also possible to provide additional command line options to the restarted server, by writing a line with `restart:` (with a colon) followed by one or more options into the expect file. These extra options will be dropped if the same server is restarted again, unless they are repeated.

4.18 Other Tips for Writing Test Cases

- Writing loops

  If you need to do something in a loop, you can use something like this:

  ```
  let $1= 10;
  while ($1)
  {
   # execute your statements here
   dec $1;
  }
  ```

- Pausing between statements

  To sleep between statements, use the `sleep` command. It supports fractions of a second. For example, `sleep 0.2;` sleeps 0.2 seconds.

  Try not to use `sleep` or `real_sleep` commands more than necessary. The more of them there are, the slower the test suite becomes. In some cases, heavy reliance on sleep operations is an indicator that the logic of a test should be reconsidered.

- Commenting the test result

  When the output in a result file is not understandable by inspection, it can be helpful to have the test case write comments to the result file that provide context. You can use the `echo` command for this:

  ```
  --echo # Comment to explain the following output
  ```

  Of course, the same line (without `--echo`) will need to be put in the corresponding place in the result file, if you are adding this to an existing test.

- Sorting result sets

  If a test case depends on `SELECT` output being in a particular row order, use an `ORDER BY` clause. Do not assume that rows will be selected in the same order they are inserted, particularly for tests that might be run multiple times under conditions that can change the order, such as with different storage engines, or with and without indexing.

If you do not want to add `ORDER BY` to the query, an alternative is to use the test command `sorted_result` which will sort the result of the following statement before putting it into the result file.

- Performing file system operations

  Avoid using `exec` or `system` to execute operating system commands for file system operations. This used to be very common, but OS commands tend to be platform specific, which reduces test portability. `mysqltest` now has several commands to perform these operations portably, so these commands should be used instead: `remove_file`, `chmod`, `mkdir`, and so forth.

- Local versus remote storage

  Some test cases depend on being run on local storage, and may fail when run on remote storage such as a network share. For example, if the test result can be affected by differences between local and remote file system times, the expected result might not be obtained. Failure of these test cases under such circumstances does not indicate an actual malfunction. It is not generally possible to determine whether tests are being run on local storage.

- Skipping consistency check or forcing restart after test

  As mentioned before, a test case should leave the server in a "clean" state for the next test. In cases where this is not possible or too costly, the test may in stead ask for the consistency check to be skipped, and the server(s) to be restarted. This is done by this executing this command at any time during the test:

  ```
  call mtr.force_restart();
  ```

  This signals to `mysql-test-run.pl` that it should restart the server before the next test, and also skip the consistency check.

Chapter 5 MySQL Test Programs

Table of Contents

This chapter describes the test programs that run test cases. For information about the language used for writing test cases, see Chapter 6, *`mysqltest` Language Reference*.

The test suite uses the following programs:

- The `mysql-test-run.pl` Perl script is the main application used to run the MySQL test suite. It invokes `mysqltest` to run individual test cases. (Prior to MySQL 4.1, a similar shell script, `mysql-test-run`, can be used instead.)
- `mysqltest` runs test cases. Prior to MySQL 8.0, a version named `mysqltest_embedded` is available; it is similar to `mysqltest` but is built with support for the `libmysqld` embedded server.
- The `mysql_client_test` program is used for testing aspects of the MySQL client API that cannot be tested using `mysqltest` and its test language. Prior to MySQL 8.0, a version named `mysql_client_test_embedded` is available; it is similar to `mysql_client_test` but is used for testing the embedded server.
- The `mysql-stress-test.pl` Perl script performs stress-testing of the MySQL server.

5.1 `mysqltest` — Program to Run Test Cases

The `mysqltest` program runs a test case against a MySQL server and optionally compares the output with a result file. This program reads input written in a special test language. Typically, you invoke `mysqltest` using `mysql-test-run.pl` rather than invoking it directly.

`mysqltest_embedded` is similar but is built with support for the `libmysqld` embedded server. This program is available only prior to MySQL 8.0.

Features of `mysqltest`:

- Can send SQL statements to MySQL servers for execution
- Can execute external shell commands
- Can test whether the result from an SQL statement or shell command is as expected
- Can connect to one or more standalone `mysqld` servers and switch between connections
- Can connect to an embedded server (`libmysqld`), if MySQL is compiled with support for `libmysqld`. (In this case, the executable is named `mysqltest_embedded` rather than `mysqltest`.)

By default, `mysqltest` reads the test case on the standard input. To run `mysqltest` this way, you normally invoke it like this:

```
shell> mysqltest [options] [db_name] < test_file
```

You can also name the test case file with a `--test-file=file_name` option.

The exit value from mysqltest is 0 for success, 1 for failure, and 62 if it skips the test case (for example, if after checking some preconditions it decides not to run the test).

mysqltest supports the following options:

- --help, -?

  Display a help message and exit.

- --basedir=*dir_name*, -b *dir_name*

  The base directory for tests.

- --character-sets-dir=*path*

  The directory where character sets are installed.

- --compress, -C

  Compress all information sent between the client and the server if both support compression.

- --cursor-protocol

  Use cursors for prepared statements.

- --database=*db_name*, -D *db_name*

  The default database to use.

- --debug[=*debug_options*], -#[*debug_options*]

  Write a debugging log if MySQL is built with debugging support. The default *debug_options* value is 'd:t:S:i:O,/tmp/mysqltest.trace'.

- --debug-check

  Print some debugging information when the program exits.

- --debug-info

  Print debugging information and memory and CPU usage statistics when the program exits.

- --explain-protocol,

  Run EXPLAIN EXTENDED on all SELECT, INSERT, REPLACE, UPDATE and DELETE queries.

- --host=*host_name*, -h *host_name*

  Connect to the MySQL server on the given host.

- --include=*file_name*, -i *file_name*

  Include the contents of the given file before processing the contents of the test file. The included file should have the same format as other mysqltest test files. This option has the same effect as putting a --source *file_name* command as the first line of the test file.

- --json-explain-protocol,

  Run EXPLAIN FORMAT=JSON on all SELECT, INSERT, REPLACE, UPDATE and DELETE queries. The json-explain-protocol option is available from MySQL 5.6.

- `--logdir=dir_name`

  The directory to use for log files.

- `--mark-progress`

  Write the line number and elapsed time to `test_file.progress`.

- `--max-connect-retries=num`

  The maximum number of connection attempts when connecting to server.

- `--max-connections=num`

  The maximum number of simultaneous server connections per client (that is, per test). If not set, the maximum is 128. Minimum allowed limit is 8, maximum is 5120.

- `--no-defaults`

  Do not read default options from any option files. If used, this must be the first option.

- `--plugin-dir=path`

  The directory in which to look for plugins. It may be necessary to specify this option if the `default_auth` argument is used for the `connect()` command to specify an authentication plugin but `mysqltest` does not find it. This option was added in MySQL 5.5.7.

- `--password[=password]`, `-p[password]`

  The password to use when connecting to the server. If you use the short option form (`-p`), you *cannot* have a space between the option and the password. If you omit the `password` value following the `--password` or `-p` option on the command line, you are prompted for one.

- `--port=port_num`, `-P port_num`

  The TCP/IP port number to use for the connection.

- `--protocol={TCP|SOCKET|PIPE|MEMORY}`

  Choose the protocol for communication with the server. `SOCKET` is default.

  The `--protocol` option is ignored if running with the embedded server.

- `--ps-protocol`

  Use the prepared-statement protocol for communication.

- `--quiet`

  Suppress all normal output. This is a synonym for `--silent`.

- `--record`, `-r`

  Record the output that results from running the test file into the file named by the `--result-file` option, if that option is given. It is an error to use this option without also using `--result-file`.

- `--result-file=file_name`, `-R file_name`

  This option specifies the file for test case expected results. `--result-file`, together with `--record`, determines how `mysqltest` treats the test actual and expected results for a test case:

  - If the test produces no results, mysqltest exits with an error message to that effect, unless --result-file is given and the named file is an empty file.
  - Otherwise, if --result-file is not given, mysqltest sends test results to the standard output.
  - With --result-file but not --record, mysqltest reads the expected results from the given file and compares them with the actual results. If the results do not match, mysqltest writes a .reject file in the same directory as the result file, outputs a diff of the two files, and exits with an error.
  - With both --result-file and --record, mysqltest updates the given file by writing the actual test results to it.

- --server-arg=*value*, -A *value*

  Pass the argument as an argument to the embedded server. For example, --server-arg=--tmpdir=/tmp or --server-arg=--core. Up to 64 arguments can be given. This option was removed in MySQL 8.0.

- --server-file=*file_name*, -F *file_name*

  Read arguments for the embedded server from the given file. The file should contain one argument per line. This option was removed in MySQL 8.0.

- --server-public-key-path=file_name

  The path name to a file containing the server RSA public key. The file must be in PEM format. The public key is used for RSA encryption of the client password for connections to the server made using accounts that authenticate with the sha256_password plugin. This option is ignored for client accounts that do not authenticate with that plugin. It is also ignored if password encryption is not needed, as is the case when the client connects to the server using an SSL connection.

  The server sends the public key to the client as needed, so it is not necessary to use this option for RSA password encryption to occur. It is more efficient to do so because then the server need not send the key.

  For additional discussion regarding use of the sha256_password plugin, including how to get the RSA public key, see SHA-256 Pluggable Authentication.

  This option is available only if MySQL was built using OpenSSL. It was added in MySQL 5.6.6 under the name --server-public-key and renamed in 5.6.7 to --server-public-key-path.

- --silent, -s

  Suppress all normal output.

- --skip-safemalloc

  Do not use memory allocation checking.

- --sleep=*num*, -T *num*

  Cause all sleep commands in the test case file to sleep *num* seconds. This option does not affect real_sleep commands.

  An option value of 0 can also be used, which effectively disables sleep commands in the test case.

- --socket=*path*, -S *path*

The socket file to use when connecting to `localhost` (which is the default host).

- `--sp-protocol`

  Execute DML statements within a stored procedure. For every DML statement, `mysqltest` creates and invokes a stored procedure that executes the statement rather than executing the statement directly.

- `--tail-lines=nn`

  Specify how many lines of the result to include in the output if the test fails because an SQL statement fails. The default is 0, meaning no lines of result printed.

- `--test-file=file_name`, `-x file_name`

  Read test input from this file. The default is to read from the standard input.

- `--timer-file=file_name`, `-m file_name`

  If given, the number of millisecond spent running the test will be written to this file. This is used by `mysql-test-run.pl` for its reporting.

- `--tls-version=protocol_list`

  The protocols permitted by the client for encrypted connections. The value is a comma-separated list containing one or more of these protocols: TLSv1, TLSv1.1, TLSv1.2. (TLSv1.2 is supported only if MySQL was compiled using OpenSSL 1.0.1 or higher. It is not supported if MySQL was compiled using yaSSL.)

  This option was added in MySQL 5.7.10.

- `--tmpdir=dir_name`, `-t dir_name`

  The temporary directory where socket files are created.

- `--trace-exec`

  If enabled, this option causes `mysqltest` to immediately display the output from executed programs to `stdout`.

  This option was added in MySQL 8.0.0.

- `--user=user_name`, `-u user_name`

  The MySQL user name to use when connecting to the server.

- `--verbose`, `-v`

  Verbose mode. Print out more information about what the program does.

- `--version`, `-V`

  Display version information and exit.

- `--view-protocol`

  Every `SELECT` statement is wrapped inside a view.

5.2 `mysql_client_test` — Test Client API

The mysql_client_test program is used for testing aspects of the MySQL client API that cannot be tested using mysqltest and its test language. mysql_client_test is run as part of the test suite.

mysql_client_test_embedded is similar but is used for testing the embedded server. This program is available only prior to MySQL 8.0.

The source code for the programs can be found in in tests/mysql_client_test.c in a source distribution. The program serves as a good source of examples illustrating how to use various features of the client API.

mysql_client_test is used in a test by the same name in the main tests suite of mysql-test-run.pl but may also be run directly. Unlike the other programs listed here, it does not read an external description of what tests to run. Instead, all tests are coded into the program, which is written to cover all aspects of the C language API.

mysql_client_test supports the following options:

- --help, -?

  Display a help message and exit.

- --basedir=*dir_name*, -b *dir_name*

  The base directory for the tests.

- --count=*count*, -t *count*

  The number of times to execute the tests.

- --database=*db_name*, -D *db_name*

  The database to use.

- --debug[=*debug_options*], -#[*debug_options*]

  Write a debugging log if MySQL is built with debugging support. The default *debug_options* value is 'd:t:o,/tmp/mysql_client_test.trace'.

- --getopt-ll-test=*option*, -g *option*

  Option to use for testing bugs in the getopt library.

- --host=*host_name*, -h *host_name*

  Connect to the MySQL server on the given host.

- --password[=*password*], -p[*password*]

  The password to use when connecting to the server. If you use the short option form (-p), you *cannot* have a space between the option and the password. If you omit the *password* value following the --password or -p option on the command line, you are prompted for one.

- --port=*port_num*, -P *port_num*

  The TCP/IP port number to use for the connection.

- --server-arg=*arg*, -A *arg*

  Argument to send to the embedded server. This option was removed in MySQL 8.0.

- `--show-tests`, `-T`

  Show all test names.

- `--silent`, `-s`

  Be more silent.

- `--socket=path`, `-S path`

  The socket file to use when connecting to `localhost` (which is the default host).

- `--testcase`, `-c`

  The option is used when called from `mysql-test-run.pl`, so that `mysql_client_test` may optionally behave in a different way than if called manually, for example by skipping some tests. Currently, there is no difference in behavior but the option is included to make this possible.

- `--user=user_name`, `-u user_name`

  The MySQL user name to use when connecting to the server.

- `-v dir_name`, `--vardir=dir_name`

  The data directory for tests. The default is `mysql-test/var`.

5.3 `mysql-test-run.pl` — Run MySQL Test Suite

Note

This content is no longer updated. Any further updates to test framework documention take place in the MySQL Source Code documentation and can be accessed at The MySQL Test Framework, Version 2.0.

The `mysql-test-run.pl` Perl script is the main application used to run the MySQL test suite. It invokes `mysqltest` to run individual test cases.

Invoke `mysql-test-run.pl` in the `mysql-test` directory like this:

```
shell> mysql-test-run.pl [options] [test_name] ...
```

Each *`test_name`* argument names a test case. The test case file that corresponds to the test name is `t/test_name.test`.

For each *`test_name`* argument, `mysql-test-run.pl` runs the named test case. With no *`test_name`* arguments, `mysql-test-run.pl` runs all `.test` files in the `t` subdirectory.

If no suffix is given for the test name, a suffix of `.test` is assumed. Any leading path name is ignored. These commands are equivalent:

```
shell> mysql-test-run.pl mytest
shell> mysql-test-run.pl mytest.test
shell> mysql-test-run.pl t/mytest.test
```

A suite name can be given as part of the test name. That is, the syntax for naming a test is:

```
[suite_name.]test_name[.suffix]
```

If a suite name is given, mysql-test-run.pl looks in that suite for the test. The test file corresponding to a test named *suite_name.test_name* is found in suite/*suite_name*/t/*test_name*.test. There is also an implicit suite name main for the tests in the top t directory. With no suite name, mysql-test-run.pl looks in the default list of suites for a match and runs the test in any suites where it finds the test. Suppose that the default suite list is main, binlog, rpl, and that a test mytest.test exists in the main and rpl suites. With an argument of mytest or mytest.test, mysql-test-run.pl will run mytest.test from the main and rpl suites.

To run a family of test cases for which the names share a common prefix, use the --do-test=*prefix* option. For example, --do-test=rpl runs the replication tests (test cases that have names beginning with rpl). --skip-test has the opposite effect of skipping test cases for which the names share a common prefix.

The argument for the --do-test and --skip-test options also allows more flexible specification of which tests to perform or skip. If the argument contains a pattern metacharacter other than a lone period, it is interpreted as a Perl regular expression and applies to test names that match the pattern. If the argument contains a lone period or does not contain any pattern metacharacters, it is interpreted the same way as previously and matches test names that begin with the argument value. For example, --do-test=testa matches tests that begin with testa, --do-test=main.testa matches tests in the main test suite that begin with testa, and --do-test=main.*testa matches test names that contain main followed by testa with anything in between. In the latter case, the pattern match is not anchored to the beginning of the test name, so it also matches names such as xmainytesta.

As of MySQL 5.7, it is possible to put a list of test names in a file and have mysql-test-run.pl run those tests, using the option --do-test-list=*file*. The tests should be listed one per line in the file, using the fully qualified name *suite.test*. A space may be used in place of the period. A line beginning with # indicates a comment and is ignored.

As of MySQL 8.0, mysql-test-run.pl supports a --do-suite option, which is similar to --do-test but permits specifying entire suites of tests to run.

To perform setup prior to running tests, mysql-test-run.pl needs to invoke mysqld with the --bootstrap and --skip-grant-tables options. If MySQL was built with the compiler flag -DDISABLE_GRANT_OPTIONS, then --bootstrap, --skip-grant-tables, and --init-file will be disabled. To handle this, set the MYSQLD_BOOTSTRAP environment variable to the full path name of a server that has all options enabled. mysql-test-run.pl will use that server to perform setup; it is not used to run the tests.

The init_file test will fail if --init-file is disabled. This is an expected failure in this case.

To run mysql-test-run.pl on Windows, you'll need either Cygwin or ActiveState Perl to run it. You may also need to install the modules required by the script. To run the test script, change location into the mysql-test directory, set the MTR_VS_CONFIG environment variable to the configuration you selected earlier (or use the --vs-config option), and invoke mysql-test-run.pl. For example (using Cygwin and the bash shell):

```
shell> cd mysql-test
shell> export MTR_VS_CONFIG=debug
shell> ./mysqltest-run.pl --force --timer
shell> ./mysqltest-run.pl --force --timer --ps-protocol
```

mysql-test-run.pl uses several environment variables. Some of them are listed in the following table. Some of these are set from the outside and used by mysql-test-run.pl, others are set by mysql-test-run.pl instead, and may be referred to in tests.

Variable	Description
MTR_BUILD_THREAD	If set, defines which port number range is used for the server
MTR_MEM	If set to anything, will run tests with files in "memory" using tmpfs or ramdisk. Not available on Windows. Same as --mem option
MTR_MAX_PARALLEL	If set, defines maximum number of parallel threads if --parallel=auto is given
MTR_*NAME*_TIMEOUT	Setting of a timeout in minutes or seconds, corresponding to command line option --*name*-timeout. Avaliable timeout names are TESTCASE, SUITE (both in minutes) and START, SHUTDOWN, CTEST (all in seconds). MTR_CTEST_TIMEOUT is for ctest unit tests; it was added in MySQL 8.0.0.
MTR_PARALLEL	If set, defines number of parallel threads executing tests. Same as --parallel option
MTR_PORT_BASE	If set, defines which port number range is used for the server
MYSQL_CONFIG_EDITOR	Path name to mysql_config_editor binary. Supported as of MySQL 5.6.6.
MYSQL_TEST	Path name to mysqltest binary
MYSQL_TEST_DIR	Full path to the mysql-test directory where tests are being run from
MYSQL_TEST_LOGIN_FILE	Path name to login file used by mysql_config_editor. If not set, the default is $HOME/.mylogin.cnf, or %APPDATA%\MySQL\.mylogin.cnf on Windows. Supported as of MySQL 5.6.6.
MYSQL_TMP_DIR	Path to temp directory used for temporary files during tests
MYSQLD	Full path to server executable used in tests. Supported as of MySQL 5.5.17.
MYSQLD_BOOTSTRAP	Full path name to mysqld that has all options enabled
MYSQLD_BOOTSTRAP_CMD	Full command line used for initial database setup for this test batch
MYSQLD_CMD	Command line for starting server as used in tests, with the minimum set of required arguments. Supported as of MySQL 5.5.17.
MYSQLTEST_VARDIR	Path name to the var directory that is used for logs, temporary files, and so forth
TSAN_OPTIONS	Path name to a file containing ThreadSanitizer suppressions. Supported as of MySQL 8.0.1.

The variable MTR_PORT_BASE is a more logical replacement for the original variable MTR_BUILD_THREAD. It gives the actual port number directly (will be rounded down to a multiple of 10). If you use MTR_BUILD_THREAD, the port number is found by multiplying this by 10 and adding 10000.

Tests sometimes rely on certain environment variables being defined. For example, certain tests assume that MYSQL_TEST is defined so that mysqltest can invoke itself with exec $MYSQL_TEST.

Other tests may refer to the last three variables listed in the preceding table, to locate files to read or write. For example, tests that need to create files will typically put them in $MYSQL_TMP_DIR/*file_name*.

The variable $MYSQLD_CMD will include any server options added with the --mysqld option to mysql-test-run.pl, but will not include server options added specifically for the currently running test.

mysql-test-run.pl supports the options in the following list. An argument of -- tells mysql-test-run.pl not to process any following arguments as options.

- `--help`, `-h`

  Display a help message and exit.

- `--big-test`

  Allow tests marked as "big" to run. Tests can be thus marked by including the line `--source include/big_test.inc`, and they will only be run if this option is given, or if the environment variable `BIG_TEST` is set to 1.

  This is typically done for tests that take very long to run, or that use very much resources, so that they are not suitable for running as part of a normal test suite run.

  If both `--big-test` and `--only-big-tests` are given, `--only-big-tests` is ignored.

- `--boot-dbx`

  Run the `mysqld` server used for bootstrapping the database through the `dbx` debugger. This option is available from MySQL 5.5.17.

- `--boot-ddd`

  Run the `mysqld` server used for bootstrapping the database through the `ddd` debugger. This option is available from MySQL 5.5.17.

- `--boot-gdb`

  Run the `mysqld` server used for bootstrapping the database through the `gdb` debugger. This option is available from MySQL 5.5.17.

  See also the `--manual-boot-gdb` option.

- `--build-thread=`*`number`*

  Specify a number to calculate port numbers from. The formula is 10 * *`build_thread`* + 10000. Instead of a number, it can be set to `auto`, which is also the default value, in which case `mysql-test-run.pl` will allocate a number unique to this host.

  The value (number or `auto`) can also be set with the `MTR_BUILD_THREAD` environment variable.

  This option is kept for backward compatibility. The more logical `--port-base` is recommended instead.

- `--callgrind`

  Instructs `valgrind` to use `callgrind`.

- `--charset-for-testdb=`*`charset_name`*

  Specify the default character set for the `test` database. The default value is `latin1`.

  This option was added in MySQL 8.0.1.

- `--check-testcases`

  Check test cases for side effects. This is done by checking the system state before and after each test case; if there is any difference, a warning to that effect is written, but the test case is not marked as failed because of it. This check is enabled by default. To disable it, use the `--nocheck-testcases` option.

- `--clean-vardir`

Clean up the var directory with logs and test results etc. after the test run, but only if there were no test failures. This option only has effect if also running with option --mem. The intent is to alleviate the problem of using up memory for test results, in cases where many different test runs are being done on the same host.

- --client-bindir=*path*

  The path to the directory where client binaries are located.

- --client-dbx

  Start mysqltest in the dbx debugger. Support for dbx is available from MySQL 5.5.12.

- --client-ddd

  Start mysqltest in the ddd debugger.

- --client-debugger=*debugger*

  Start mysqltest in the named debugger.

- --client-gdb

  Start mysqltest in the gdb debugger.

- --client-libdir=*path*

  The path to the directory where client libraries are located.

- --combination=*value*

  Extra option to pass to mysqld. The value should consist of a single mysqld option including dashes. This option is similar to --mysqld but has a different effect. mysql-test-run.pl executes multiple test runs, using the options for each instance of --combination in successive runs. If --combination is given only once, it has no effect. For test runs specific to a given test suite, an alternative to the use of --combination is to create a combinations file in the suite directory. The file should contain a section of options for each test run. See Section 4.9, “Passing Options from mysql-test-run.pl to mysqld or mysqltest”.

- --comment=*str*

  Write *str* to the output within lines filled with #, as a form of banner.

- --compress

  Compress all information sent between the client and the server if both support compression.

- --cursor-protocol

  Pass the --cursor-protocol option to mysqltest (implies --ps-protocol).

- --dbx

  Start mysqld in the dbx debugger. Support for dbx is available from MySQL 5.5.12.

- --ddd

  Start mysqld in the ddd debugger.

- `--debug`

  Dump trace output for all clients and servers.

- `--debugger=debugger`

  Start `mysqld` using the named debugger.

- `--debug-common`

  This option works similar to `--debug` but turns on debug only for the debug macro keywords `query, info, error, enter, exit` which are considered the most commonly used.

- `--debug-server`

  Runs `mysqld.debug` (if available) instead of `mysqld` as server. If it does find `mysqld.debug`, it will search for plugin libraries in a subdirectory `debug` under the directory where it's normally located. This option does not turn on trace output and is independent of the `debug` option.

- `--debug-sync-timeout=seconds`

  Controls whether the Debug Sync facility for testing and debugging is enabled. The option value is a timeout in seconds. The default value is 300. A value of 0 disables Debug Sync. The value of this option also becomes the default timeout for individual synchronization points.

  `mysql-test-run.pl` passes `--loose-debug-sync-timeout=seconds` to `mysqld`. The `--loose` prefix is used so that `mysqld` does not fail if Debug Sync is not compiled in.

  For information about using the Debug Sync facility for testing, see Section 4.15, “Thread Synchronization in Test Cases”.

- `--default-myisam`

  Use `MyISAM` as the default storage engine for all except `InnoDB`-specific tests. This option is on by default in MySQL 5.5 and 5.6, but is off by default as of MySQL 5.7. See also `--nodefault-myisam`.

- `--defaults-file=file_name`

  Use the named file as fixed config file template for all tests.

- `--defaults_extra_file=file_name`

  Add setting from the named file to all generated configs.

- `--discover`

  Attempt to preload `discover`, the Developer Studio Memory Error Discovery Tool when starting `mysqld`. Reports from `discover` may be found in `log/mysqld.%p.txt` under the directory given by `--vardir`. This option was added in MySQL 8.0.1. It is supported only on SPARC-M7 systems.

- `--do-suite=prefix or regex`

  Run all test cases from suites having a name that begins with the given `prefix` value or matches the regular expression. If the argument matches no existing suites, `mysql-test-run.pl` aborts.

  The argument for the `--do-suite` option allows more flexible specification of which tests to perform. See the description of the `--do-test` option for details.

  The `--do-suite` option was added in MySQL 8.0.

- `--do-test=prefix or regex`

  Run all test cases having a name that begins with the given *`prefix`* value or matches the regular expression. This option provides a convenient way to run a family of similarly named tests.

  The argument for the `--do-test` option allows more flexible specification of which tests to perform. If the argument contains a pattern metacharacter other than a lone period, it is interpreted as a Perl regular expression and applies to test names that match the pattern. If the argument contains a lone period or does not contain any pattern metacharacters, it is interpreted the same way as previously and matches test names that begin with the argument value. For example, `--do-test=testa` matches tests that begin with `testa`, `--do-test=main.testa` matches tests in the `main` test suite that begin with `testa`, and `--do-test=main.*testa` matches test names that contain `main` followed by `testa` with anything in between. In the latter case, the pattern match is not anchored to the beginning of the test name, so it also matches names such as `xmainytestz`.

- `--do-testlist=file`

  Run all tests listed in the file *`file`*. In this file, tests should be listed one per line in the form *`suite.test`* or alternatively, with a space instead of the period. A line beginning with `#` will be ignored and can be used for comments.

  The `--do-test-list` option is available from MySQL 5.7.

- `--embedded-server`

  Use a version of `mysqltest` built with the embedded server. This option was removed in MySQL 8.0.

- `--enable-disabled`

  Ignore any `disabled.def` file, and run also tests marked as disbaled. Success or failure of those tests will be reported the same way as other tests.

- `--experimental=file_name`

  Specify a file that contains a list of test cases that should be displayed with the `[ exp-fail ]` code rather than `[ fail ]` if they fail.

  For an example of a file that might be specified using this option, see `mysql-test/collections/default.experimental`.

  It is also possible to supply more than one `--experimental`, test cases listed in all the files will be treated as experimental.

- `--explain-protocol`,

  Run `EXPLAIN EXTENDED` on all `SELECT`, `INSERT`, `REPLACE`, `UPDATE`, and `DELETE` statements.

- `--extern option=value`

  Use an already running server. The option/value pair is what is needed by the `mysql` client to connect to the server. Each `--extern` can only take one option/value pair as argument, so it you need more you need to repeat `--extern` for each of them. Example:

  ```
  ./mysql-test-run.pl --extern socket=var/tmp/mysqld.1.sock alias
  ```

  Note: If a test case has an `.opt` file that requires the server to be restarted with specific options, the file will not be used. The test case likely will fail as a result.

- `--fast`

  Do not perform controlled shutdown when servers need to be restarted or at the end of the test run. This is equivalent to using `--shutdown-timeout=0`.

- `--fail-check-testcases`

  Enabling this option when a test is run, causes it to fail if MTR's internal check of the test case fails. If this option is disabled, only a warning is generated while the test passes. This option is enabled by default. For additional information, see the description of the `--check-testcases` option.

  The `--fail-check-testcases` option was added in MySQL 8.0.

- `--force`

  Normally, `mysql-test-run.pl` exits if a test case fails. `--force` causes execution to continue regardless of test case failure.

- `--force-restart`

  Always restart the server(s) between each tast case, whether it's needed or not. Will also restart between repeated runs of the same test case. This may be useful e.g. when looking for the source of a memory leak, as there will only have been one test run before the server exits.

- `--gcov`

  Run tests with the `gcov` test coverage tool.

- `--gdb`

  Start `mysqld` in the `gdb` debugger.

- `--gprof`

  Run tests with the `gprof` profiling tool.

- `--include-ndbcluster`, `--include-ndb`

  Run also tests that need Cluster.

- `--json-explain-protocol`,

  Run `EXPLAIN FORMAT=JSON` on all SELECT, INSERT, REPLACE, UPDATE and DELETE queries. The `json-explain-protocol` option is available from MySQL 5.6.

- `--manual-boot-gdb`

  This option is similar to `--boot-gdb` but attaches the debugger to the server during the bootstrapping process, permitting the use of a remote debugger. This option is available from MySQL 5.7.14.

- `--manual-dbx`

  Use a server that has already been started by the user in the `dbx` debugger. Support for dbx is available from MySQL 5.5.12.

- `--manual-ddd`

  Use a server that has already been started by the user in the `ddd` debugger.

- `--manual-debug`

Use a server that has already been started by the user in a debugger.

- `--manual-gdb`

  Use a server that has already been started by the user in the `gdb` debugger.

- `--mark-progress`

  Marks progress with timing (in milliseconds) and line number in `var/log/testname.progress`.

- `--max-connections=num`

  The maximum number of simultaneous server connections that may be used per test. If not set, the maximum is 128. Minimum allowed limit is 8, maximum is 5120. Corresponds to the same option for `mysqltest`.

- `--max-save-core=N`

  Limit the number of core files saved, to avoid filling up disks in case of a frequently crashing server. Defaults to 5, set to 0 for no limit. May also be set with the environment variable `MTR_MAX_SAVE_CORE`

- `--max-save-datadir=N`

  Limit the number of data directories saved after failed tests, to avoid filling up disks in case of frequent failures. Defaults to 20, set to 0 for no limit. May also be set with the environment variable `MTR_MAX_SAVE_DATADIR`

- `--max-test-fail=N`

  Stop execution after the specified number of tests have failed, to avoid using up resources (and time) in case of massive failures. retries are noe counted, nor are failures of tests marked experimental. Defaults to 10, set to 0 for no limit. May also be set with the environment variable `MTR_MAX_TEST_FAIL`

- `--mem`

  This option is not supported on Windows.

  Run the test suite in memory, using tmpfs or ramdisk. This can decrease test times significantly, in particular if you would otherwise be running over a remote file system. `mysql-test-run.pl` attempts to find a suitable location using a built-in list of standard locations for tmpfs and puts the `var` directory there. This option also affects placement of temporary files, which are created in `var/tmp`.

  The default list includes `/dev/shm`. You can also enable this option by setting the environment variable `MTR_MEM[=dir_name]`. If *`dir_name`* is given, it is added to the beginning of the list of locations to search, so it takes precedence over any built-in locations.

  Once you have run tests with `--mem` within a `mysql-test`directory, a soflink `var` will have been set up to the temporary directory, and this will be re-used the next time, until the soflink is deleted. Thus, you do not have to repeat the `--mem` option next time.

- `--mysqld=value`

  Extra option to pass to `mysqld`. Only one option may be specified in *`value`*; to specify more than one, use additional `--mysqld` options. See Section 4.9, “Passing Options from `mysql-test-run.pl` to `mysqld` or `mysqltest`”.

- `--mysqld-env=variable=value`

Sets (or changes) an environment variable before starting mysqld. Varibles set in the environment from which you run mysql-test-run.pl will normally also be propagated to mysqld, but there may be cases where you want a setting just for a single run, or you may not want the setting to affect other programs. You may use additional --mysqld-env options to set more than one variable.

- --mysqltest=*options*

  Extra options to pass to mysqltest.

  This option was added in MySQL 8.0.0.

- --ndb-connectstring=*str*

  Pass --ndb-connectstring=*str* to the master MySQL server. This option also prevents mysql-test-run.pl from starting a cluster. It is assumed that there is already a cluster running to which the server can connect with the given connectstring.

- --nocheck-testcases

  Disable the check for test case side effects. For additional information, see the description of the --check-testcases option.

- --nodefault-myisam

  For MySQL 5.5 or 5.6, do not override the build-in default engine to use MyISAM instead for non-InnoDB tests. Since the existing collection of tests were originally adapted for MyISAM as default, many tests will fail when this option is used, because the test behaves differently or produces different output when the engine switches to InnoDB.

  From MySQL 5.7, the default engine for tests has been changed to InnoDB and this option will have no effect.

- --noreorder

  Do not reorder tests to reduce number of restarts, but run them in exactly the order given. If a whole suite is to be run, the tests are run in alphabetic order, though similiar combinations will be grouped together. If more than one suite is listed, the tests are run one suite at a time, in the order listed.

- --no-skip

  This option forces all tests to run, ignoring any --skip commands used in the test. This ensures that all tests are run. An excluded list (excludenoskip.list) is maintained to track which tests should continue to be skipped. The --no-skip option continues to skip the tests that are named in the excluded list. The default value of --no-skip introduced variable is OFF, which implies users are not forced to run all tests unless the --no-skip is explicitly used.

  ```
  shell> mysql-test-run.pl
           --suite=innodb
           --no-skip
  ```

- --notimer

  Cause mysqltest not to generate a timing file. The effect of this is that the report from each test case does not include the timing in milliseconds as it normally does.

- --nounit-tests

Do not run unit tests, overriding default behavior or setting of the MTR_UNIT_TESTS variable.

Running of unit tests was enabled from MySQL 5.5.11.

- --nowarnings

  Do not look for and report errors and warning in the server logs.

- --only-big-tests

  This option causes only big tests to run. Normal (non-big) tests are skipped. If both --big-test and --only-big-tests are given, --only-big-tests is ignored.

  --only-big-tests was added in MySQL 8.0.1.

- --parallel={*N*|auto}

  Run tests using *N* parallel threads. By default, 1 thread is used. Use --parallel=auto to set *N* automatically.

  Setting the MTR_PARALLEL environment variable to *N* has the same effect as specifying --parallel=*N*.

  The MTR_MAX_PARALLEL environment variable, if set, specifies the maximum number of parallel workers that can be spawned when the --parallel=auto option is specified. If --parallel=auto is not specified, MTR_MAX_PARALLEL variable has no effect.

- --port-base=*P*

  Specify base of port numbers to be used; a block of 10 will be allocated. *P* should be divisible by 10; if it is not, it will be rounded down. If running with more than one parallel test thread, thread 2 will use the next block of 10 and so on.

  If the port number is given as auto, which is also the default, mysql-test-run.plwill allocate a number unique to this host. The value may also be given with the environment variable MTR_PORT_BASE.

  --port-base was added in MySQL 5.1.45 as a more logical alternative to --build-thread. If both are used, --port-base takes precedence.

- --print-testcases

  Do not run any tests, but print details about all tests, in the order they would have been run.

- --ps-protocol

  Pass the --ps-protocol option to mysqltest.

- --record

  Pass the --record option to mysqltest. This option requires a specific test case to be named on the command line.

- --reorder

  Reorder tests to minimize the number of server restarts needed. This is the default behavior. There is no guarantee that a particular set of tests will always end up in the same order.

- `--repeat=N`

  Run each test *N* number of times.

- `--report-features`

  Display the output of `SHOW ENGINES` and `SHOW VARIABLES`. This can be used to verify that binaries are built with all required features.

- `--report-times`

  At the end of the test run, write a summary of how much time was spent in various phases of execution. If you run with `--parallel`, the total will exceed the wall clock time passed, since it will be summed over all threads.

  The times reported should only be treated as approximations, and the exact points where the time is taken may also change between releases. If the test run is aborted, including if a test fails and `--force` is not in use, the time report will not be produced.

  The `--report-times` is available from MySQL 5.5.

- `--retry=N`

  If a test fails, it is retried up to a maximum of *N* runs, but will terminate after 2 failures. Default is 3, set to 1 or 0 for no retries. This option has no effect unless `--force` is also used; without it, test execution will terminate after the first failure.

  The `--retry` and `--retry-failure` options do not affect how many times a test repeated with `--repeat` may fail in total, as each repetition is considered a new test case, which may in turn be retried if it fails.

- `--retry-failure=N`

  Allow a failed and retried test to fail more than the default 2 times before giving it up. Setting it to 0 or 1 effectively turns off retries

- `--sanitize`

  Scan the server log files for warnings from various sanitizers. Use of this option assumes that MySQL was configured with `-DWITH_ASAN` or `-DWITH_UBSAN`.

  This option was added in MySQL 8.0.0. As of MySQL 8.0.1, the `TSAN_OPTIONS` environment variable can be set to specify the path name of a file containing ThreadSanitizer suppressions.

- `--shutdown-timeout=seconds`

  Max number of seconds to wait for servers to do controlled shutdown before killing them. Default is 10.

- `--skip-combinations`

  Do not apply combinations; ignore combinations file or option.

- `--skip-ndbcluster`, `--skip-ndb`

  Do not start NDB Cluster; skip Cluster test cases. This option only has effect if you do have NDB, if not it will have no effect as it cannot run those tests anyway.

- `--skip-ndbcluster-slave`, `--skip-ndb-slave`

Do not start an NDB Cluster slave.

- `--skip-rpl`

  Skip replication test cases.

- `--skip-ssl`

  Do not start `mysqld` with support for SSL connections.

- `--skip-test=regex`

  Specify a regular expression to be applied to test case names. Cases with names that match the expression are skipped. tests to skip.

  The argument for the `--skip-test` option allows more flexible specification of which tests to skip. If the argument contains a pattern metacharacter other than a lone period, it is interpreted as a Perl regular expression and applies to test names that match the pattern. See the description of the `--do-test` option for details.

- `--skip-test-list=file`

  Specify a file listing tests that should be skipped (disabled).

  The file has the same format as the `disabled.def` file listing disabled tests. With this option, disabling can be done on a case by case basis. The `--skip-test-list` option is supported from MySQL 5.5.

- `--skip-*`

  `--skip-*` options not otherwise recognized by `mysql-test-run.pl` are passed to the master server.

- `--sleep=N`

  Pass `--sleep=N` to `mysqltest`.

- `--sp-protocol`

  Pass the `--sp-protocol` option to `mysqltest`.

- `--ssl`

  If `mysql-test-run.pl` is started with the `--ssl` option, it sets up a secure connection for all test cases. In this case, if `mysqld` does not support SSL, `mysql-test-run.pl` exits with an error message: `Couldn't find support for SSL`

- `--start`

  Initialize and start servers with the startup settings for the specified test case. You can use this option to start a server to which you can connect later. For example, after building a source distribution you can start a server and connect to it with the `mysql` client like this:

```
shell> cd mysql-test
shell> ./mysql-test-run.pl --start alias &
shell> ../mysql -S ./var/tmp/master.sock -h localhost -u root
```

  If no tests are named on the command line, the server(s) will be started with settings for the first test that would have been run without the `--start` option.

mysql-test-run.pl will stop once the server has been started, but will terminate if the server dies. If killed, it will also shut down the server.

- --start-and-exit

  This is similar to --start, but mysql-test-run.pl terminates once the server has been started, leaving just the server process running.

- --start-dirty

  This is similar to --start, but will skip the database initialization phase and assume that database files are already available. Usually this means you must have run another test first.

- --start-from=*test_name*

  mysql-test-run.pl sorts the list of names of the test cases to be run, and then begins with *test_name*.

- --strace-client

  Create strace output for mysqltest. Will produce default strace output as mysqltest.strace. Note that this will be overwritten for each new test case, so it's most useful for running only one test.

  The strace-client option is functional from MySQL 5.5.20, and only supported on Linux. The option was available in earlier versions too, but was not working properly.

- --strace-server

  Create strace output for the server. Will produce default strace output as mysqld.1.strace. Note that this will be overwritten each time the server is restarted, so it's most useful for running a single test, or if you want trace from the first test that fails.

  The strace-server option is available from MySQL 5.5.20, on Linux only.

- --stress=*stress options*

  Start a server, but instead of running a test, run mysql-stress-test.pl with the supplied arguments. Arguments needed to communicate with the server will be automatically provided, the rest should be given as arguments to this option. Command line options for mysql-stress-test.pl should be separeted by a comma.

  The stress option was added in MySQL 5.5.17, it is not a direct replacement for the option of the same name that exists in version 1 of mysql-test-run.pl.

- --suite=*suite_name*

  Run the named test suite. The default name is main (the regular test suite located in the mysql-test directory).

- --suite-timeout=*minutes*

  Specify the maximum test suite runtime in minutes.

- --summary-report=*file_name*

  Generate a plain text version of the test summary only and write it to the file named as the option argument. The file is suitable for sending by email. This option was added in MySQL 8.0.1.

- `--test-progress`

  Display the percentage of tests remaining. This option was added in MySQL 5.7.19.

- `--testcase-timeout=`*`minutes`*

  Specify the maximum test case runtime in minutes.

- `--timediff`

  Adds to each test report for a test case, the total time in sconds and milliseconds passed since the preceding test ended. This option can only be used together with `--timestamp`, and has no effect without it.

- `--timer`

  Cause `mysqltest` to generate a timing file. The default file is named `./var/log/timer`.

- `--timestamp`

  Prints a timestamp before the test case name in each test report line, showing when the test ended.

- `--tmpdir=`*`path`*

  The directory where temporary file are stored. The default location is `./var/tmp`. The environment variable `MYSQL_TMP_DIR` will be set to the path for this directory, whether it has the default value or has been set explicitly. This may be referred to in tests.

- `--unit-tests`

  Force running of unit tests, overriding default behavior or setting of the `MTR_UNIT_TESTS` variable.

  Running of unit tests was enabled from MySQL 5.5.11.

- `--unit-tests-report`

  Extend the unit test run by also outputting the log from the test run, independently of whether it succeeded or not. This option implies `--unit-tests` so it is not necessary to specify both. The `--unit-tests-report` option is available in MySQL 5.5 from version 5.5.44, in 5.6 from version 5.6.25 as well as in MySQL 5.7.

- `--user=`*`user_name`*

  The MySQL user name to use when connecting to the server.

- `--user-args`

  Drops all non-essential command line arguments to the `mysqld` server, except those supplied with `--mysqld` arguemnts, if any. Only works in combination with `--start`, `--start-and-exit` or `--start-dirty`, and only if no test name is given.

- `--valgrind`

  Run `mysqltest` and `mysqld` with `valgrind`. This and the following `--valgrind` options require that the executables have been build with `valgrind` support.

  When the server is run with valgrind, an extra pass over the server log file(s) will be performed after all tests are run, and any report with problems that have been reported at server shutdown will be extracted

and printed. The most common warnings are memory leaks. With each report will also be listed all tests that were run since previous server restart; one of these is likely to have caused the problem.

From MySQL 5.5.13, a final "pseudo" test named `valgrind_report` is added to the list of tests when the server is run in valgrind. This test is reported as failed if any such shutdown warnings were produced by valgrind. Pass or failure of this test is also added to the total test count reported.

- `--valgrind-clients`

  Run all clients started by `.test` files with `valgrind`. This option requires `valgrind` 3.9 or later.

  `--valgrind-clients` was added in MySQL 5.7.9.

- `--valgrind-mysqld`

  Run the `mysqld` server with `valgrind`.

- `--valgrind-mysqltest`

  Run `mysqltest` with `valgrind`.

- `--valgrind-option=`*`str`*

  Extra options to pass to `valgrind`.

- `--valgrind-path=`*`path`*

  Specify the path name to the `valgrind` executable.

- `--vardir=`*`path`*

  Specify the path where files generated during the test run are stored. The default location is `./var`. The environment variable `MYSQLTEST_VARDIR` will be set to the path for this directory, whether it has the default value or has been set explicitly. This may be referred to in tests.

- `--verbose`

  Give more verbose output regarding test execution. Use the option twice to get even more output. Note that the output generated within each test case is not affected.

- `--verbose-restart`

  Write when and why servers are restarted between test cases.

- `--view-protocol`

  Pass the `--view-protocol` option to `mysqltest`.

- `--vs-config=`*`config_val`*

  Specify the configuration used to build MySQL (for example, `--vs-config=debug --vs-config=release`). This option is for Windows only.

- `--wait-all`

  If `--start` or `--start-dirty` is used, wait for all servers to exit before termination. Otherise, it will terminate if one (of several) servers is restarted.

- `--warnings`

Search the server log for errors or warning after each test and report any suspicious ones; if any are found, the test will be marked as failed. This is the default behavior, it may be turned off with `--nowarnings`.

- `--with-ndbcluster-only`

  Run only test cases that have `ndb` in their name.

Note

The hostname resolves to 127.0.0.1 and not to the actual IP address.

5.4 `mysql-stress-test.pl` — Server Stress Test Program

The `mysql-stress-test.pl` Perl script performs stress-testing of the MySQL server.

`mysql-stress-test.pl` requires a version of Perl that has been built with threads support.

Invoke `mysql-stress-test.pl` like this:

```
shell> mysql-stress-test.pl [options]
```

`mysql-stress-test.pl` supports the following options:

- `--help`

  Display a help message and exit.

- `--abort-on-error=N`

  Causes the program to abort if an error with severity less than or equal to N was encountered. Set to 1 to abort on any error.

- `--check-tests-file`

  Periodically check the file that lists the tests to be run. If it has been modified, reread the file. This can be useful if you update the list of tests to be run during a stress test.

- `--cleanup`

  Force cleanup of the working directory.

- `--log-error-details`

  Log error details in the global error log file.

- `--loop-count=N`

  In sequential test mode, the number of loops to execute before exiting.

- `--mysqltest=path`

  The path name to the `mysqltest` program.

- `--server-database=db_name`

  The database to use for the tests. The default is `test`.

- `--server-host=host_name`

  The host name of the local host to use for making a TCP/IP connection to the local server. By default, the connection is made to `localhost` using a Unix socket file.

- `--server-logs-dir=path`

  This option is required. *`path`* is the directory where all client session logs will be stored. Usually this is the shared directory that is associated with the server used for testing.

- `--server-password=password`

  The password to use when connecting to the server.

- `--server-port=port_num`

  The TCP/IP port number to use for connecting to the server. The default is 3306.

- `--server-socket=file_name`

  For connections to `localhost`, the Unix socket file to use, or, on Windows, the name of the named pipe to use. The default if `/tmp/mysql.sock`.

- `--server-user=user_name`

  The MySQL user name to use when connecting to the server. The default is `root`.

- `--sleep-time=N`

  The delay in seconds between test executions.

- `--stress-basedir=path`

  This option is required. *`path`* is the working directory for the test run. It is used as the temporary location for result tracking during testing.

- `--stress-datadir=path`

  The directory of data files to be used during testing. The default location is the `data` directory under the location given by the `--stress-suite-basedir` option.

- `--stress-init-file[=path]`

  `file_name` is the location of the file that contains the list of tests to be run once to initialize the database for the testing. If missing, the default file is `stress_init.txt` in the test suite directory.

- `--stress-mode=mode`

  This option indicates the test order in stress-test mode. The *`mode`* value is either `random` to select tests in random order or `seq` to run tests in each thread in the order specified in the test list file. The default mode is `random`.

- `--stress-suite-basedir=path`

  This option is required. *`path`* is the directory that has the `t` and *`r`* subdirectories containing the test case and result files. This directory is also the default location of the `stress-test.txt` file that contains the list of tests. (A different location can be specified with the `--stress-tests-file` option.)

- `--stress-tests-file[=file_name]`

Use this option to run the stress tests. *file_name* is the location of the file that contains the list of tests. If *file_name* is omitted, the default file is stress-test.txt in the stress suite directory. (See --stress-suite-basedir.)

- --suite=*suite_name*

  Run the named test suite. The default name is main (the regular test suite located in the mysql-test directory).

- --test-count=*N*

  The number of tests to execute before exiting.

- --test-duration=*N*

  The duration of stress testing in seconds.

- --threads=*N*

  The number of threads. The default is 1.

- --verbose

  Verbose mode. Print more information about what the program does.

Chapter 6 mysqltest Language Reference

Table of Contents

This chapter describes the test language implemented by `mysqltest`. The language allows input to contain a mix of comments, commands executed by `mysqltest` itself, and SQL statements that `mysqltest` sends to a MySQL server for execution.

Terminology notes:

- A “command” is an input test that `mysqltest` recognizes and executes itself. A “statement” is an SQL statement or query that `mysqltest` sends to the MySQL server to be executed.
- When `mysqltest` starts, it opens a connection it calls `default` to the MySQL server, using any connection parameters specified by the command options. (For a local server, the default user name is `root`. For an external server, the default user name is `test` or the user specified with the `--user` option.) You can use the `connect` command to open other connections, the `connection` command to switch between connections, and the `disconnect` command to close connections. However, the capability for switching connections means that the connection named `default` need not be the connection in use at a given time. To avoid ambiguity, this document avoids the term “default connection.” It uses the term “current connection” to mean “the connection currently in use,” which might be different from “the connection named `default`.”

6.1 mysqltest Input Conventions

`mysqltest` reads input lines and processes them as follows:

- “End of line” means a newline (linefeed) character. A carriage return/linefeed (CRLF) pair also is allowable as as a line terminator (the carriage return is ignored). Carriage return by itself is *not* allowed as a line terminator.
- A line that begins with “`#`” as the first nonwhitespace content is treated as a comment that extends to the end of the line and is ignored. Example:

```
# this is a comment
```

- Earlier versions would also allow comments beginning with “`--`” unless the first word was a valid `mysqltest` command, but this has been deprecated and is longer allowed.
- Other input is taken as normal command input. The command extends to the next occurrence of the command delimiter, which is semicolon (“`;`”) by default. The delimiter can be changed with the `delimiter` command.

  If `mysqltest` recognizes the first word of the delimiter-terminated command, `mysqltest` executes the command itself. Otherwise, `mysqltest` assumes that the command is an SQL statement and sends it to the MySQL server to be executed.

Because the command extends to the delimiter, a given input line can contain multiple commands, and a given command can span multiple lines. The ability to write multiple-line statements is useful for making long statements more readable, such as a `create table` statement for a table that has many columns.

After `mysqltest` reads a command up to a delimiter and executes it, input reading restarts following the delimiter and any remaining input on the line that contains the delimiter is treated as though it begins on a new line. Consider the following two input lines:

```
echo issue a select statement; select 1; echo done
issuing the select statement;
```

That input contains two commands and one SQL statement:

```
echo issue a SELECT statement
SELECT 1;
echo done issuing the SELECT statement
```

Similarly, "`#`" comments can begin on a command line following a delimiter:

```
SELECT 'hello'; # select a string value
```

On a multiple-line command, "`#`" or "`--`" at the beginning of the second or following lines is not special. Thus, the second and third lines of the following variable-assignment command are not taken as comments. Instead, the variable `$a` is set to a value that contains two linefeed characters:

```
let $a= This is a variable
# assignment that sets a variable
-- to a multiple-line value;
```

`--` commands and normal commands have complementary properties with regard to how `mysqltest` reads them:

- A "`--`" command is terminated by a newline, regardless of how many delimiters it contains.
- A normal command (without "`--`") is terminated by the delimiter (semicolon), no matter how many newlines it contains.

`mysqltest` commands can be written either with a leading "`--`") or as normal command input (no leading "`--`"). Use the command delimiter only in the latter case. Thus, these two lines are equivalent:

```
--sleep 2
sleep 2;
```

The equivalence is true even for the `delimiter` command. For example, to set the delimiter to "`//`", either of these commands work:

```
--delimiter //
delimiter //;
```

To set the delimiter back to ";", use either of these commands:

```
--delimiter ;
delimiter ;//
```

A potential ambiguity occurs because a command line can contain either a mysqltest command or an SQL statement. This has a couple of implications:

- No mysqltest command should be the same as any keyword that can begin an SQL statement.
- Should extensions to SQL be implemented in the future, it is possible that a new SQL keyword could be impossible for mysqltest to recognize as such if that keyword is already used as a mysqltest command.

Any ambiguity can be resolved by using the "--" syntax to force interpetation as a mysqltest command, or the query command to force interpretation as SQL.

All file paths used in test commands should use forward slash "/" as the directory separator as in Unix. They will be automatically converted when needed if the test is run on Windows. We also recommend putting all temporary or auxiliary files made during the test under the directory referred to by $MYSQL_TMP_DIR. Do not put them under fixed full paths like /tmp. This will help ensuring portability of the test, and avoiding conflicts with other programs.

$MYSQL_TMP_DIR is equivalent to $MYSQLTEST_VARDIR/tmp if you are not running with parallel test threads, but if you run mysql-test-run.pl with --parallel, they will be different. It is therefore best to be consistent and use $MYSQL_TMP_DIR.

From MySQL 5.5.17, commands named disable_X or enable_X, except parsing, reconnect and rpl_parse, can take an optional modifier ONCE. If this is added, the relevant setting is enabled or disabled only for the next command or statement, after which it is reverted to whatever it was before. Note that the word ONCE must be in upper case; this was chosen in order to make it more visible when reading the test script.

For example, --disable_query_log ONCE will ensure query log is disabled for the next statement, but will not affect whether or not query log is enabled for statements following the next. It is possible to enable/disable more that one property (e.g. both query log and result log) for a single statement using the ONCE modifier.

6.2 mysqltest Commands

mysqltest supports the commands described in this section. Command names are not case sensitive.

Some examples of command use are given, but you can find many more by searching the test case files in the mysql-test/t directory.

- append_file *file_name* [*terminator*]

  append_file is like write_file except that the lines up to the terminator are added to the end of the file. The file is created if it does not exist. The file name argument is subject to variable substitution.

```
write_file $MYSQL_TMP_DIR/data01;
line one for the file
line two for the file
EOF
append_file $MYSQL_TMP_DIR/data01;
line three for the file
EOF
```

```
write_file $MYSQL_TMP_DIR/data02 END_OF_FILE;
line one for the file
line two for the file
```

```
END_OF_FILE
append_file $MYSQL_TMP_DIR/data02 END_OF_FILE;
line three for the file
END_OF_FILE
```

- `cat_file file_name`

  `cat_file` writes the contents of the file to the output. The file name argument is subject to variable substitution.

```
cat_file $MYSQL_TMP_DIR/data01;
```

- `change_user [user_name], [password], [db_name]`

  Changes the current user and causes the database specified by *db_name* to become the default database for the current connection.

```
change_user root;
--change_user root,,test
```

- `character_set charset_name`

  Set the default character set to *charset_name*. Initially, the character set is `latin1`.

```
character_set utf8;
--character_set sjis
```

- `chmod octal_mode file_name`

  Change the mode of the given file. The file mode must be given as a four-digit octal number. The file name argument is subject to variable substitution, but must evaluate to a literal file name, not a file name pattern.

```
chmod 0644 $MYSQL_TMP_DIR/data_xxx01;
```

- `connect (name, host_name, user_name, password, db_name [,port_num [,socket [,options [,default_auth]]]])`

  Open a connection to the server and make the connection the current connection.

  The arguments to `connect` are:

  - *name* is the name for the connection (for use with the `connection`, `disconnect`, and `dirty_close` commands). This name must not already be in use by an open connection.
  - *host_name* indicates the host where the server is running.
  - *user_name* and *password* are the user name and password of the MySQL account to use.
  - *db_name* is the default database to use. As a special case, `*NO-ONE*` means that no default database should be selected. You can also leave *db_name* blank to select no database.
  - *port_num*, if given, is the TCP/IP port number to use for the connection. This parameter can be given by using a variable.
  - *socket*, if given, is the socket file to use for connections to `localhost`. This parameter can be given by using a variable.

- *options* can be one or more of the following words, separated by spaces:
  - CLEARTEXT: Enable use of the cleartext authentication plugin. This option was added in MySQL 5.5.27.
  - COMPRESS: Use the compressed client/server protocol, if available.
  - PIPE: Use the named-pipe connection protocol, if available.
  - SHM: Use the shared-memory connection protocol, if available.
  - SOCKET: Use the socket-file connection protocol. This option was added in MySQL 8.0.1.
  - TCP: Use the TCP/IP connection protocol. This option was added in MySQL 8.0.1.
  - SSL: Use a secure connection.

  Prior to MySQL 8.0.1, passing PIPE or SHM on non-Windows systems caused the default (socket-file) connection protocol to be used. As of 8.0.1, this causes an error, and, similarly, passing SOCKET on Windows systems causes an error.
- *default_auth* is the name of an authentication plugin. It is passed to the mysql_options() C API function using the MYSQL_DEFAULT_AUTH option. If mysqltest does not find the plugin, use the --plugin-dir option to specify the directory where the plugin is located.

  This argument can be used as of MySQL 5.5.7.

To omit an argument, just leave it blank. For an omitted argument, mysqltest uses an empty string for the first five arguments and the *options* argument. For omitted port or socket options, mysqltest uses the default port or socket.

```
connect (conn1,localhost,root,,);
connect (conn2,localhost,root,mypass,test);
connect (conn1,127.0.0.1,root,,test,$MASTER_MYPORT);
```

The last example assumes that the $MASTER_MYPORT variable has already been set (perhaps as an environment variable).

If a connection attempt fails initially, mysqltest retries five times if the abort-on-error setting is enabled.

- connection *connection_name*

  Select *connection_name* as the current connection. To select the connection that mysqltest opens when it starts, use the name default.

```
connection master;
connection conn2;
connection default;
```

  A variable can be used to specify the *connection_name* value.

- let $var= convert_error(*error*)

  This is not a command as such but rather a function that can be used in let statements. If the argument is a number, it returns the name of the corresponding error, or <Unknown> if no such error exists. If the

argument is an error name, it returns the corresponding number, or fails if the error name is unknown. If the argument is 0 or an empty string, it returns 0. The function can also take a variable as argument.

```
let $errvar1=convert_error(ER_UNKNOWN_ERROR);
let $errvar2=convert_error(1450);
let $errvar3=convert_error($errvar1);
```

The `convert_error` function was added in MySQL 5.6.

- `copy_file from_file to_file`

  Copy the file *`from_file`* to the file *`to_file`*. The command fails if *`to_file`* already exists. The file name arguments are subject to variable substitution.

- `copy_files_wildcard src_dir_name dst_dir_name [pattern]`

  Copy all files that match the pattern in the source directory to the destination directory. Patterns can use `?` to represent any single character, or `*` for any sequence of 0 or more characters. The `.` character is treated like any other. The pattern may not include `/`.

  The command works like this:

  - Files that match the pattern are copied from the source directory to the destination directory. Overwriting of files is permitted.
  - Copying does not apply to directories matching the pattern or matching files in subdirectories.
  - If the source or destination directory is not present, an error occurs.
  - The pattern argument is optional. If no pattern is provided, all files from the source directory are copied to the destination directory.
  - If a pattern is provided, but no files match it, an error occurs.
  - If the source directory has no files, an error occurs.

  ```
  copy_files_wildcard $MYSQLTEST_VARDIR/std_data/ $MYSQLTEST_VARDIR/copy1/ *.txt
  ```

- `dec $var_name`

  Decrement a numeric variable. If the variable does not have a numeric value, the result is undefined.

  ```
  dec $count;
  dec $2;
  ```

- `delimiter str`

  Set the command delimiter to *`str`*, which may consist of 1 to 15 characters. The default delimiter is the semicolon character (“`;`”).

  ```
  delimiter /;
  --delimiter stop
  ```

  This is useful or needed when you want to include long SQL statements like `CREATE PROCEDURE` which include semicolon delimited statements but need to be interpreted as a single statement by `mysqltest`. If you have set the delimiter to “`/`” as in the previous example, you can set it back to the default like this:

```
delimiter ;|
```

- `die [message]`

  Aborts the test with an error code after printing the given message as the reason. Suppose that a test file contains the following line:

  ```
  die Cannot continue;
  ```

  When `mysqltest` encounters that line, it produces the following result and exits:

  ```
  mysqltest: At line 1: Cannot continue
  not ok
  ```

- `diff_files file_name1 file_name2`

  Compare the two files. The command succeeds if the files are the same, and fails if they are different or either file does not exist. The file name arguments are subject to variable substitution.

- `dirty_close connection_name`

  Close the named connection. This is like `disconnect` except that it calls `vio_delete()` before it closes the connection. If the connection is the current connection, you should use the `connection` command to switch to a different connection before executing further SQL statements.

  A variable can be used to specify the *`connection_name`* value.

- `disable_abort_on_error`, `enable_abort_on_error`

  Disable or enable abort-on-error behavior. This setting is enabled by default. With this setting enabled, `mysqltest` aborts the test when a statement sent to the server results in an unexpected error, and does not generate the `.reject` file. For discussion of reasons why it can be useful to disable this behavior, see Section 6.5, “Error Handling”.

  ```
  --disable_abort_on_error
  --enable_abort_on_error
  ```

- `disable_connect_log`, `enable_connect_log`

  Disable or enable logging of creation or switch of connections. Connection logging is disabled by default. With this setting enabled, `mysqltest` enters lines in the test results to show when connections are created, switched or disconnected.

  If query logging is turned off using `disable_query_log`, connection logging is also turned off, until query log is re-enabled.

  ```
  --disable_connect_log
  --enable_connect_log
  ```

- `disable_info`, `enable_info`

  Disable or enable additional information about SQL statement results. Information display is disabled by default. With this setting enabled, `mysqltest` displays the affected-rows count and the output from

the mysql_info() C API function. The “affected-rows” value is “rows selected” for statements such as SELECT and “rows modified” for statements that change data.

```
--disable_info
--enable_info
```

- disable_metadata, enable_metadata

  Disable or enable query metadata display. Metadata display is disabled by default. With this setting enabled, mysqltest adds query metadata to the result. This information consists of the values corresponding to the members of the MYSQL_FIELD C API data structure, for each column of the result.

  ```
  --disable_metadata
  --enable_metadata
  ```

- disable_parsing, enable_parsing

  Disable or enable query parsing. This setting is enabled by default. When disabled, mysqltest ignores everything until enable_parsing. These commands are useful for "commenting out" a section from a test case without having to add a comment marker to every single line.

  ```
  --disable_parsing
  --enable_parsing
  ```

- disable_ps_protocol, enable_ps_protocol

  Disable or enable prepared-statement protocol. This setting is disabled by default unless the --ps-protocol option is given.

  ```
  --disable_ps_protocol
  --enable_ps_protocol
  ```

- disable_query_log, enable_query_log

  Disable or enable query logging. This setting is enabled by default. With this setting enabled, mysqltest echoes input SQL statements to the test result.

  One reason to disable query logging is to reduce the amount of test output produced, which also makes comparison of actual and expected results more efficient.

  ```
  --disable_query_log
  --enable_query_log
  ```

- disable_reconnect, enable_reconnect

  Disable or enable automatic reconnect for dropped connections. (The default depends on the client library version.) This command only applies to the current connection.

  ```
  --disable_reconnect
  --enable_reconnect
  ```

- disable_result_log, enable_result_log

  Disable or enable the result log. This setting is enabled by default. With this setting enabled, mysqltest displays query results (and results from commands such as echo and exec).

```
--disable_result_log
--enable_result_log
```

- `disable_rpl_parse`, `enable_rpl_parse`

  Disable or enable parsing of statements to determine whether they go to the master or slave. The default is whatever the default is for the C API library.

  ```
  --disable_rpl_parse
  --enable_rpl_parse
  ```

- `disable_session_track_info`, `enable_session_track_info`

  Disable or enable display of session tracking information. These commands were added in MySQL 5.7. Session-tracking display disabled by default.

  ```
  --disable_session_track_info
  --enable_session_track_info
  ```

- `disable_warnings`, `enable_warnings`

  Disable or enable warnings. This setting is enabled by default. With this setting enabled, `mysqltest` uses `SHOW WARNINGS` to display any warnings produced by SQL statements.

  ```
  --disable_warnings
  --enable_warnings
  ```

- `disconnect` *`connection_name`*

  Close the named connection. If the connection is the current connection, you should use the `connection` command to switch to a different connection before executing further SQL statements.

  ```
  disconnect conn2;
  disconnect slave;
  ```

- `echo` *`text`*

  Echo the text to the test result. References to variables within the text are replaced with the corresponding values. The text does not need to be enclosed in quotation marks; if it is, the quotation marks will be included in the output.

  ```
  --echo Another sql_mode test
  echo should return only 1 row;
  ```

- `end`

  End an `if` or `while` block. If there is no such block open, `mysqltest` exits with an error. See Section 6.4, “`mysqltest` Flow Control Constructs”, for information on flow-control constructs.

  `mysqltest` considers `}` and `end` the same: Both end the current block.

- `end_timer`

  Stop the timer. By default, the timer does not stop until just before `mysqltest` exits.

- `error error_code [, error_code] ...`

  Specify one or more comma-separated error values that the next command is expected to return. Each *`error_code`* value is a MySQL-specific error number or an SQLSTATE value. (These are the kinds of values returned by the `mysql_errno()` and `mysql_sqlstate()` C API functions, respectively.)

  If you specify an SQLSTATE value, it should begin with an `S` to enable `mysqltest` to distinguish it from a MySQL error number. For example, the error number 1050 and the SQLSTATE value `42S01` are equivalent, so the following commands specify the same expected error:

  ```
  --error 1050
  --error S42S01
  ```

  SQLSTATE values should be five characters long and may contain only digits and uppercase letters.

  It is also possible to use the symbolic error name from `mysqld_error.h`:

  ```
  --error ER_TABLE_EXISTS_ERROR
  ```

  As of MySQL 8.0, it is also possible to use symbolic error names from `errmsg.h` to refer to client errors:

  ```
  --error CR_SERVER_GONE_ERROR
  ```

  Finally, you can assign either a numerical code or a symbolic error name to a variable and refer to that in the `error` command. This feature was added in MySQL 5.5.18. Numbers, symbolic names and variables may be freely mixed.

  If a statement fails with an error that has not been specified as expected by means of a `error` command, `mysqltest` aborts and reports the error message returned by the MySQL server.

  If a statement fails with an error that has been specified as expected by means of a `error` command, `mysqltest` does not abort. Instead, it continues and writes a message to the result output.

  - If an `error` command is given with a single error value and the statement fails with that error, `mysqltest` reports the error message returned by the MySQL server.

    Input:

    ```
    --error S42S02
    DROP TABLE t;
    ```

    `mysqltest` reports:

    ```
    ERROR 42S02: Unknown table 't'
    ```

  - If an `error` command is given with multiple error values and the statement fails with any of those errors, `mysqltest` reports a generic message. (This is true even if the error values are all the same, a fact that can be used if you want a message that does not contain varying information such as table names.)

    Input:

    ```
    --error S41S01,S42S02
    DROP TABLE t;
    ```

mysqltest reports:

```
Got one of the listed errors
```

An error value of 0 or S00000 means “no error,” so using either for an error command is the same as saying explicitly, “no error is expected, the statement must succeed.”.

To indicate that you expect success or a given error or errors, specify 0 or S00000 first in the error list. If you put the no-error value later in the list, the test will abort if the statement is successful. That is, the following two commands have different effects: The second form literally means the next command may fail with error code 0, (rather than succeed) which in practice never happens:

```
--error 0,1051
--error 1051,0
```

You can use error to specify shell status values for testing the value of shell commands executed using the exec command. This does not apply to system, for which the command status is ignored.

If you use error in combination with send and reap, the error should be used just before the reap, as this is the command that actually gives the result and the potential error.

From MySQL 5.5.18, variables may also be used as arguments to the error command; these may contain a number (including 0), an SQLSTATE or a symbolic error name. Variables and constant values may be freely combined.

- eval *statement*

  Evaluate the statement by replacing references to variables within the text with the corresponding values. Then send the resulting statement to the server to be executed. Use “\$” to specify a literal “$” character.

  The advantage of using eval *statement* versus just *statement* is that eval provides variable expansion.

  ```
  eval USE $DB;
  eval CHANGE MASTER TO MASTER_PORT=$SLAVE_MYPORT;
  eval PREPARE STMT1 FROM "$my_stmt";
  ```

- exec *command* [*arg*] ...

  Execute the shell command using the popen() library call. References to variables within the command are replaced with the corresponding values. Use “\$” to specify a literal “$” character.

  On Cygwin, the command is executed from cmd.exe, so commands such as rm cannot be executed with exec. Use system instead.

  ```
  --exec $MYSQL_DUMP --xml --skip-create test
  --exec rm $MYSQLTEST_VARDIR/tmp/t1
  exec $MYSQL_SHOW test -v -v;
  ```

Note

exec or system are sometimes used to perform file system operations, but the command for doing so tend to be operating system specific, which reduces test

portability. mysqltest now has several commands to perform these operations portably, so they should be used instead: remove_file, chmod, mkdir, and so forth.

- execw *command* [*arg*] ...

  This is a variant of the exec command which is needed on Windows if the command line contains non-ASCII characters. Otherwise it works exactly the same. On platforms other than Windows there is no difference, but on Windows it uses a different version of the popen() library call. So if your command line contains non-ASCII characters, it is recommended to use execw instead of exec.

  The execw command is available from MySQL 5.6.

- exit

  Terminate the test case. This is considered a “normal termination.” That is, using exit does not result in evaluation of the test case as having failed. It is not necessary to use exit at the end of a test case, as the test case will terminate normally when reaching the end without failure.

- expr $*var_name*= *operand1* *operator* *operand2*

  Evaluate an expression and assign the result to a variable. The result is also the return value of the expr command itself.

```
--let $val1= 10
--let $var2= 20
--expr $res= $var1 + $var2
--echo $res
```

  operand1 and *operand2* must be valid variables.

  expr supports these mathematical operators:

```
+  Addition
-  Subtraction
*  Multiplication
/  Division
%  Modulo
```

  expr supports these logical operators:

```
&&  Logical AND
||  Logical OR
```

  expr supports these bitwise operators:

```
&  Binary AND
|  Binary OR
^  Binary XOR
<< Binary left shift
>> Binary right shift
```

  Operations are subject to these conditions:

  - Operations that do not support noninteger operands truncate such operands to integer values.
  - If the result is an infinite value, expr returns the inf keyword.

- Division by 0 results in an infinite value.

The expr command was added in MySQL 8.0.1.

- file_exists *file_name*

file_exists succeeds if the named file exists and fails otherwise. The file name argument is subject to variable substitution.

```
file_exists /etc/passwd;
```

- force-cpdir *src_dir_name dst_dir_name*

Copies the source directory, *src_dir_name*, to the destination directory, *dst_dir_name*. The copy operation is recursive, so it copies subdirectories. Returns 0 for success and 1 for failure.

```
--force-cpdir testdir testdir2
```

If the source directory does not exist, an error occurs.

If the destination directory does not exist, mysqltest creates it before copying the source directory.

force-cpdir was added in MySQL 8.0.1.

- force-rmdir *dir_name*

Remove a directory named *dir_name*. Returns 0 for success and 1 for failure.

```
--force-rmdir testdir
```

force-rmdir removes the directory as well as its contents, if any, unlike rmdir, which fails if the directory to be removed contains any files or directories.

force-rmdir was added in MySQL 8.0.

- horizontal_results

Set the default query result display format to horizontal. Initially, the default is to display results horizontally.

```
--horizontal_results
```

- if (*expr*)

Begin an if block, which continues until an end or } line. mysqltest executes the block if the expression is non-zero. There is no provision for else with if. See Section 6.4, “mysqltest Flow Control Constructs”, for further information about if statements.

```
let $counter= 0;
if ($counter)
{
  echo Counter is not 0;
}
if (!$counter)
{
```

```
  echo Counter is 0;
}
```

- `inc $var_name`

  Increment a numeric variable. If the variable does not have a numeric value, the result is undefined.

```
inc $i;
inc $3;
```

- `let $var_name = value`

  `let $var_name = query_get_value(query, col_name, row_num)`

  Assign a value to a variable. The variable name cannot contain whitespace or the "=" character. Except for the one-digit `$0` to `$9`, it cannot begin with a number. `mysqltest` aborts with an error if the value is erroneous.

  References to variables within *`value`* are replaced with their corresponding values.

  If the `let` command is specified as a normal command (that is, not beginning with "`--`"), *`value`* includes everything up to the command delimiter, and thus can span multiple lines.

```
--let $1= 0
let $count= 10;
```

  When assigning a literal string to a variable, no quoting is required even if the string contains spaces. If the string does include quotation marks, they will be trated like any other characters and be included in the string value. This is important to be aware of when using the variable in an SQL statement.

  The result from executing a query can be assigned to a variable by enclosing the query within backtick ("`` ` ``") characters:

```
let $q= `SELECT VERSION()`;
```

  The `let` command can set environment variables, not just `mysqltest` test language variables. To assign a value to an environment variable rather than a test language variable, just omit the dollar sign:

```
let $mysqltest_variable= foo;
let ENV_VARIABLE= bar;
```

  This is useful in interaction with external tools. In particular, when using the `perl` command, the Perl code cannot access test language variables, but it can access environment variables. For example, the following statement can access the `ENV_VARIABLE` value:

```
print $ENV{'ENV_VARIABLE'};
```

  The `let` syntax also allows the retrieval of a value from a query result set produced by a statement such as `SELECT` or `SHOW`. See the description of `query_get_value()` for more information.

- `mkdir dir_name`

  Create a directory named *`dir_name`*. Returns 0 for success and 1 for failure.

```
--mkdir testdir
```

- `list_files dir_name [pattern]`

  `list_files` lists the files in the named directory. If a pattern is given, lists only file(s) matching the pattern, which may contain wild cards.

```
--list_files $MYSQLD_DATADIR/test t1*
```

- `list_files_append_file file_name dir_name [pattern]`

  `list_files_append_file` works like `list_files`, but rather than outputting the file list, it is appended to the file named in the first argument. If the file does not exist, it is created.

```
--list_files_append_file $MYSQL_TMP_DIR/filelist $MYSQL_TMP_DIR/testdir *.txt;
```

- `list_files_write_file file_name dir_name [pattern]`

  `list_files_write_file` works like `list_files_append_file`, but creates a new file to write the file list to. If the file already exists, it will be replaced.

```
--list_files_write_file $MYSQL_TMP_DIR/filelist $MYSQL_TMP_DIR/testdir *.txt;
```

- `lowercase_result`

  Output from the following SQL statement will be converted to lowercase. This is sometimes needed to ensure consistent result across different platforms. If this is combined with one of the `replace` commands or with `sorted_result`, both will take effect on the output, with conversion to lowercase being applied first.

```
--lowercase_result
```

- `move_file from_name to_name`

  `move_file` renames *`from_name`* to *`to_name`*. The file name arguments are subject to variable substitution, but must evaluate to a literal file name, not a file name pattern.

```
move_file $MYSQL_TMP_DIR/data01 $MYSQL_TMP_DIR/test.out;
```

- `output file_name`

  Direct output from the next SQL statement to the named file rather than to the test output. If the file already exists, it will be overwritten. Only the next SQL statement will have its output redirected. This command is available from MySQL 5.7.

```
output $MYSQL_TMP_DIR/out-file
```

- `perl [terminator]`

  Use Perl to execute the following lines of the test file. The lines end when a line containing the terminator is encountered. The default terminator is `EOF`, but a different terminator can be provided.

```
perl;
print "This is a test\n";
EOF
```

```
perl END_OF_FILE;
print "This is another test\n";
END_OF_FILE
```

- `ping`

  Ping the server. This executes the `mysql_ping()` C API function. The function result is discarded. The effect is that if the connection has dropped and reconnect is enabled, pinging the server causes a reconnect.

- `query [statement]`

  Send the statement to the server to be executed. The `query` command can be used to force `mysqltest` to send a statement to the server even if it begins with a keyword that is a `mysqltest` command.

- `query_get_value(query, col_name, row_num)`

  The `query_get_value()` function can be used only on the right hand side of a variable assigment in a `let` statement.

  `query_get_value()` enables retrieval of a value from a query result set produced by a statement such as `SELECT` or `SHOW`. The first argument indicates the query to execute. The second and third arguments indicate the column name and row number that specify which value to extract from the result set. The column name is case sensitive. Row numbers begin with 1. The arguments can be given literally or supplied using variables.

  Suppose that the test file contains this input:

  ```
  CREATE TABLE t1(a INT, b VARCHAR(255), c DATETIME);
  SHOW COLUMNS FROM t1;
  let $value= query_get_value(SHOW COLUMNS FROM t1, Type, 1);
  echo $value;
  ```

  The result will be:

  ```
  CREATE TABLE t1(a INT, b VARCHAR(255), c DATETIME);
  SHOW COLUMNS FROM t1;
  Field   Type    Null    Key     Default Extra
  a       int(11) YES             NULL
  b       varchar(255)    YES             NULL
  c       datetime        YES             NULL
  int(11)
  ```

  If the query fails, an error message occurs and the test fails.

- `query_horizontal statement`

  Execute the statement and display its result horizontally.

  ```
  query_horizontal SELECT PI();
  ```

- `query_vertical statement`

  Execute the statement and display its result vertically.

  ```
  ```

```
query_vertical SELECT PI();
```

- `real_sleep num`

  Sleep *num* seconds. *num* can have a fractional part. Unlike the `sleep` command, `real_sleep` is not affected by the `--sleep` command-line option.

  ```
  --real_sleep 10
  real_sleep 5;
  ```

  Try not to use `sleep` or `real_sleep` commands more than necessary. The more of them there are, the slower the test suite becomes.

- `reap`

  Receive the result of the statement sent with the `send` command within the current session. You should not use *reap* unless a statement has been sent with `send`, and you should not use `send` again if there is an outstanding `send` that has not been processed with `reap`.

- `remove_file file_name`

  `remove_file` removes the file. It fails with an error if the file does not exist. The file name argument is subject to variable substitution, but must evaluate to a literal file name, not a file name pattern.

  ```
  remove_file $MYSQL_TMP_DIR/data01;
  ```

- `remove_files_wildcard dir_name [pattern]`

  Remove all files in the named directory that match the pattern. Removal does not apply to directories matching the pattern or matching files in subdirectories. Patterns can use `?` to represent any single character, or `*` for any sequence of 0 or more characters. The `.` character is treated like any other. The pattern may not include `/`.

  If no pattern argument is given, all files in the directory will be removed, but not the directory itself.

  ```
  remove_files_wildcard $MYSQL_TMP_DIR file*.txt;
  ```

- `replace_column col_num value [col_num value] ...`

  Replace strings in the output from the next statement. The value in *col_num* is replaced by the corresponding *value*. There can be more than one *col_num*/*value* pair. Column numbers start with 1.

  A replacement value can be double-quoted. (Use "`\"`" to specify a double quote within a replacement string.) Variables can be used in a replacement value if it is not double-quoted.

  If mixed `replace_xxx` commands are given, only the final one applies.

  

  Note

  Although `replace_regex` and `replace_result` affect the output from `exec`, `replace_column` does not because `exec` output is not necessarily columnar.

  ```
  --replace_column 9 #
  replace_column 1 b 2 d;
  ```

- `replace_regex /pattern/replacement/[i] ...`

In the output from the next statement, find strings within columns of the result set that match *pattern* (a regular expression) and replace them with *replacement*. Each instance of a string in a column that matches the pattern is replaced. Matching is case sensitive by default. Specify the optional i modifier to cause matching to be case insensitive.

The syntax for allowable patterns is the same as for the REGEXP SQL operator. In addition, the pattern can contain parentheses to mark substrings matched by parts of the pattern. These substrings can be referenced in the replacement string: An instance of \N in the replacement string causes insertion of the *N*-th substring matched by the pattern. For example, the following command matches strawberry and replaces it with raspberry and strawberry:

```
--replace_regex /(strawberry)/raspberry and \1/
```

Multiple *pattern*/*replacement* pairs may be given. The following command replaces instances of A with C (the first pattern replaces A with B, the second replaces B with C):

```
--replace_regex /A/B/ /B/C/
```

If a given pattern is not found, no error occurs and the input is unchanged.

- replace_result *from_val to_val* [*from_val to_val*] ...

  Replace strings in the result. Each occurrence of *from_val* is replaced by the corresponding *to_val*. There can be more than *from_val*/*to_val* pair. Arguments can be quoted with single quotation marks or double quotation marks. Variable references within the arguments are expanded before replacement occurs. Values are matched literally. To use patterns, use the replace_regex command.

  ```
  --replace_result 1024 MAX_KEY_LENGTH 3072 MAX_KEY_LENGTH
  replace_result $MASTER_MYPORT MASTER_PORT;
  ```

- require *file_name*

  This command specifies a file to be used for comparison against the results of the next query. If the contents of the file do not match or there is some other error, the test aborts with a “this test is not supported” error message.

  ```
  --require r/slave-stopped.result
  --require r/have_moscow_leap_timezone.require
  ```

  As of MySQL 8.0.1, require command is removed.

- reset_connection

  Reset the connection state by calling mysql_reset_connection(). This command is available from MySQL 5.7.

- result *file_name*

  This command specifies a file to be used for comparison when the test case completes. If the content does not match or there is some other error, write the result to r/*file_name*.reject.

  If the --record command-line option is given, the result command changes the file by writing the new test result to it.

- result_format *version*

Set the format to the specified version, which is either 1 for the current, default behavior, or to 2 which is an extended alternative format. The setting is in effect until it is changed or until the end of the test.

In format version 2, empty lines and indentation in the test file are preserveded in the result. Also, comments indicated by a double ## are copied verbatim to the result. Comments using a single # are not copied. Format version 2 makes it easier for humans to read the result output, but at the cost of somewhat larger files due to the white space and comments.

```
--result_format 2
```

- `rmdir dir_name`

  Remove a directory named *`dir_name`*. Returns 0 for success and 1 for failure.

  ```
  --rmdir testdir
  ```

  `rmdir` fails if the directory to be removed contains any files or directories. To remove the directory as well as its contents, if any, use `force-rmdir`.

- `save_master_pos`

  For a master replication server, save the current binary log file name and position. These values can be used for subsequent `sync_with_master` or `sync_slave_with_master` commands.

- `send [statement]`

  Send a statement to the server but do not wait for the result. The result must be received with the `reap` command. You cannot execute another SQL statement on the same connection between `send` and `reap`.

  If *`statement`* is omitted, the `send` command applies to the next statement executed. This means that `send` can be used on a line by itself before a statement. Thus, this command:

  ```
  send SELECT 1;
  ```

  Is equivalent to these commands:

  ```
  send;
  SELECT 1;
  ```

- `send_eval [statement]`

  Evaluate the command, then send it to the server. This is a combination of the `send` and `eval` commands, giving the functionality of both. After variable replacement has been done, it behaves like the `send` command.

  ```
  --send_eval $my_stmt
  ```

- `send_quit connection`

  Sends a COM_QUIT command to the server on the named connection.

  ```
  send_quit con;
  ```

- `send_shutdown`

  Sends a shutdown command to the server but does not wait for it to complete the shutdown. Test execution proceeds as soon as the shutdown command is sent.

- `shutdown_server [timeout]`

  Stops the server. This command waits for the server to shut down by monitoring its process ID (PID) file. If the server's process ID file is not gone after *`timeout`* seconds, the process will be killed. If *`timeout`* is omitted, the default is 60 seconds.

```
shutdown_server;
shutdown_server 30;
```

- `skip [message]`

  Skips the rest of the test file after printing the given message as the reason. This can be used after checking a condition that must be satisfied, as a way of performing an exit that displays a reason. Suppose that the test file `mytest` has these contents:

```
let $v= 0;
if (!$v)
{
  skip value is zero, skipping test;
}
echo This command is never reached;
```

  Executing `mysqltest -x mytest` yields these results:

```
The test './mytest' is not supported by this installation
Detected in file ./mytest at line 4
reason: value is zero, skipping test
```

  If the test is run from `mysql-test-run.pl`, you will instead see the test result as `[ skipped ]` followed by the message.

- `sleep num`

  Sleep *`num`* seconds. *`num`* can have a fractional part. If the `--sleep` command-line option was given, the option value overrides the value given in the `sleep` command. For example, if `mysqltest` is started with `--sleep=10`, the command `sleep 15` sleeps 10 seconds, not 15.

```
--sleep 10
sleep 0.5;
```

  Try not to use `sleep` or `real_sleep` commands more than necessary. The more of them there are, the slower the test suite becomes.

- `sorted_result`

  Sort the output from the next statement if it produces a result set. `sorted_result` is applied just before displaying the result, after any other result modifiers that might have been specified, such as `replace_result` or `replace_column`. If the next statement produces no result set, `sorted_result` has no effect because there is nothing to sort.

```
sorted_result;
```

```
SELECT 2 AS "my_col" UNION SELECT 1;
let $my_stmt=SELECT 2 AS "my_col" UNION SELECT 1;
--sorted_result
eval $my_stmt;
--sorted_result
--replace_column 1 #
SELECT '1' AS "my_col1",2 AS "my_col2"
UNION
SELECT '2',1;
```

`sorted_result` sorts the entire result of the next query. If this involves constructs such as `UNION`, stored procedures, or multi-statements, the output will be in a fixed order, but all the results will be sorted together and might appear somewhat strange.

The purpose of the `sorted_result` command is to produce output with a deterministic order for a given set of result rows. It is possible to use `ORDER BY` to sort query results, but that can sometimes present its own problems. For example, if the optimizer is being investigated for some bug, `ORDER BY` might order the result but return an incorrect set of rows. `sorted_result` can be used to produce sorted output even in the absence of `ORDER BY`.

`sorted_result` is useful for eliminating differences between test runs that may otherwise be difficult to compensate for. Results without `ORDER BY` are not guaranteed to be returned in any given order, so the result for a given query might differ between test runs. For example, the order might vary between different server versions, so a result file created by one server might fail when compared to the result created by another server. The same is true for different storage engines. `sorted_result` eliminates these order differences by producing a deterministic row order.

Other ways to eliminate differences from results without use of `sorted_result` include:

- Remove columns from the select list to reduce variability in the output
- Use aggregate functions such as `AVG()` on all columns of the select list
- Use `ORDER BY`

The use of aggregate functions or `ORDER BY` may also have the advantage of exposing other bugs by introducing additional stress on the server. The choice of whether to use `sorted_result` or `ORDER BY` (or perhaps both) may be dictated by whether you are trying to expose bugs, or avoid having them affect results. This means that care should be taken with `sorted_result` because it has the potential of hiding server bugs that result in true problems with result order.

- `source` *`file_name`*

  Read test input from the named file.

  If you find that several test case files contain a common section of commands (for example, statements that create a standard set of tables), you can put those commands in another file and those test cases that need the file can include it by means of a `source` *`file_name`* command. This enables you to write the code just once rather than in multiple test cases.

  Normally, the file name in the `source` command is relative to the `mysql-test` directory because `mysqltest` usually is invoked in that directory.

  A sourced file can use `source` to read other files, but take care to avoid a loop. The maximum nesting level is 16.

  ```
  --source include/have_csv.inc
  ```

```
source include/varchar.inc;
```

The file name can include variable references. Variables are expanded including any quotation marks in the values, so normally the values should not include quotation marks. Suppose that `/tmp/junk` contains this line:

```
SELECT 'I am a query';
```

The following example shows one way in which variable references could be used to specify the file name:

```
let $dir= /tmp;
let $file= junk;
source $dir/$file;
```

- `start_timer`

  Restart the timer, overriding any timer start that occurred earlier. By default, the timer starts when `mysqltest` begins execution.

- `sync_slave_with_master [connection_name]`

  Executing this command is equivalent to executing the following commands:

  ```
  save_master_pos;
  connection connection_name;
  sync_with_master 0;
  ```

  If *`connection_name`* is not specified, the connection named `slave` is used.

  The effect is to save the replication coordinates (binary log file name and position) for the server on the current connection (which is assumed to be a master replication server), and then switch to a slave server and wait until it catches up with the saved coordinates. Note that this command implicitly changes the current connection.

  A variable can be used to specify the *`connection_name`* value.

- `sync_with_master offset`

  For a slave replication server, wait until it has caught up with the master. The position to synchronize to is the position saved by the most recent `save_master_pos` command plus *`offset`*.

  To use this command, `save_master_pos` must have been executed at some point earlier in the test case to cause `mysqltest` to save the master's replication coordinates.

- `system command [arg] ...`

  Execute the shell command using the `system()` library call. References to variables within the command are replaced with the corresponding values. Use “`\$`” to specify a literal “`$`” character.

  ```
  --system echo '[mysqltest1]' > $MYSQLTEST_VARDIR/tmp/tmp.cnf
  --system echo 'port=1234' >> $MYSQLTEST_VARDIR/tmp/tmp.cnf
  system rm $MYSQLTEST_VARDIR/master-data/test/t1.MYI;
  ```

Note

`exec` or `system` are sometimes used to perform file system operations, but the command for doing so tend to be operating system specific, which reduces test portability. `mysqltest` now has several commands to perform these operations portably, so they should be used instead: `remove_file`, `chmod`, `mkdir`, and so forth.

- `vertical_results`

  Set the default query result display format to vertical. Initially, the default is to display results horizontally.

```
--vertical_results
```

- `wait_for_slave_to_stop`

  Poll the current connection, which is assumed to be a connection to a slave replication server, by executing `SHOW STATUS LIKE 'Slave_running'` statements until the result is `OFF`.

  For information about alternative means of slave server control, see Section 4.14, “Writing Replication Tests”.

- `while (expr)`

  Begin a `while` loop block, which continues until an `end` line. `mysqltest` executes the block repeatedly as long as the expression is true (non-zero). See flow-control constructs. Section 6.4, “`mysqltest` Flow Control Constructs”, for further information about `while` statements.

  Make sure that the loop includes some exit condition that eventually occurs. This can be done by writing *`expr`* so that it becomes false at some point.

```
let $i=5;
while ($i)
{
  echo $i;
  dec $i;
}
```

- `write_file file_name [terminator]`

  Write the following lines of the test file to the given file, until a line containing the terminator is encountered. The default terminator is `EOF`, but a different terminator can be provided. The file name argument is subject to variable substitution. An error occurs if the file already exists.

```
write_file $MYSQL_TMP_DIR/data01;
line one for the file
line two for the file
EOF
```

```
write_file $MYSQL_TMP_DIR/data02 END_OF_FILE;
line one for the file
line two for the file
END_OF_FILE
```

6.3 mysqltest Variables

You can define variables and refer to their values. You can also refer to environment variables, and there is a built-in variable that contains the result of the most recent SQL statement.

To define a variable, use the `let` command. Examples:

```
let $a= 14;
let $b= this is a string;
--let $a= 14
--let $b= this is a string
```

The variable name cannot contain whitespace or the "`=`" character.

If a variable has a numeric value, you can increment or decrement the value:

```
inc $a;
dec $a;
--inc $a
--dec $a
```

`inc` and `dec` are commonly used in `while` loops to modify the value of a counter variable that controls loop execution.

The result from executing a query can be assigned to a variable by enclosing the query within backtick ("`` ` ``") characters:

```
let $q= `select version()`;
```

References to variables can occur in the `echo`, `eval`, `exec`, and `system` commands. Variable references are replaced by their values. A nonquery value assigned to a variable in a `let` command can also refer to variables.

Variable references that occur within `` `query` `` are expanded before the query is sent to the server for execution.

You can refer to environment variables. For example, this command displays the value of the `$PATH` variable from the environment:

```
--echo $PATH
```

`$mysql_errno` is a built-in variable that contains the numeric error returned by the most recent SQL statement sent to the server, or 0 if the statement executed successfully. `$mysql_errno` has a value of −1 if no statement has yet been sent.

From MySQL 5.5.17, `$mysql_errname` similarly contains the symbolic name of the last error, or an empty string if there was no error.

`mysqltest` first checks `mysqltest` variables and then environment variables. `mysqltest` variable names are not case sensitive. Environment variable names are case sensitive.

6.4 mysqltest Flow Control Constructs

The syntax for `if` and `while` blocks looks like this:

```
if (expr)
{
  command list
}
```

```
while (expr)
{
  command list
}
```

An expression result is true if nonzero, false if zero. If the expression begins with `!`, the sense of the test is reversed.

If the expression is a string that does not begin with a numeric digit (possibly preceeded by a plus or minus sign), it evaluates as true if non-empty. Any white space is ignored in this case, so a string consisting of only white space is false.

There is no provision for `else` with `if`.

For a `while` loop, make sure that the loop includes some exit condition that eventually occurs. This can be done by writing *`expr`* so that it becomes false at some point.

The allowable syntax for *`expr`* is *`$var_name`*, *`!$var_name`*, a string or integer, or *`` `query` ``*.

From MySQL 5.5, the expression can also be a simple comparison, where the left hand side must be a variable, and the right hand side can be any type valid for the single expression except the negated variable. The supported operators are `==`, `!=`, `<`, `<=`, `>` and `>=`. Only the first two may be used if the right hand side does not evaluate to an integer. With `==`, strings must match exactly.

If you use a string on the right hand side of the comparison, it does not have to be quoted even if it contains spaces. It may optionally be enclosed in single or double quotation marks which will then be stripped off before comparison. This is in contrast to `let` statements, where quoting is not stripped. The optional quoting is not available in the first release of MySQL 5.5 GA (5.5.8), but can be used from 5.5.9.

Examples of the expression syntax with comparisons (only the header shown):

```
while ($counter<5) ...
if ($value == 'No such row') ...
if ($slave_count != $master_count) ...
```

The opening `{` (curly brace) must be separated from the preceding `)` (right parenthesis) by whitespace, such as a space or a line break.

Variable references that occur within *`` `query` ``* are expanded before the query is sent to the server for execution.

6.5 Error Handling

If an expected error is specified and that error occurs, `mysqltest` continues reading input. If the command is successful or a different error occurs, `mysqltest` aborts.

If no expected error is specified, `mysqltest` aborts unless the command is successful. (It is implicit that you expect `$mysql_errno` to be 0.)

By default, `mysqltest` aborts for certain conditions:

- A statement that fails when it should have succeeded. The following statement should succeed if table `t` exists;

```
SELECT * FROM t;
```

- A statement that fails with an error different from that specified:

```
--error 1
SELECT * FROM no_such_table;
```

- A statement that succeeds when an error was expected:

```
--error 1
SELECT 'a string';
```

You can disable the abort for errors of the first type by using the `disable_abort_on_error` command. In this case, when errors occur for statements that should succeed, `mysqltest` continues processing intput.

`disable_abort_on_error` does *not* cause `mysqltest` to ignore errors for the other two types, where you explicitly state which error you expect. This behavior is intentional. The rationale is that if you use the `error` command to specify an expected error, it is assumed that the test is sufficiently well characterized that only the specified error is accceptable.

If you do not use the `error` command, it is assumed that you might not know which error to expect or that it might be difficult to characterize all possible errors that could occur. In this case, `disable_abort_on_error` is useful for causing `mysqltest` to continue processing input. This can be helpful in the following circumstances:

- During test case development, it is useful to process all input even if errors occur so that you can see all errors at once, such as those that occur due to typographical or syntax errors. Otherwise, you can see and fix only one scripting problem at a time.
- Within a file that is included with a `source` command by several different test cases, errors might vary depending on the processing environment that is set up prior to the `source` command.
- Tests that follow a given statement that can fail are independent of that statement and do not depend on its result.

Chapter 7 Creating and Executing Unit Tests

Table of Contents

Storage engines and plugins can have unit tests to test their components. The top-level `Makefile` target `test-unit` runs all unit tests: It scans the storage engine and plugin directories, recursively, and executes all executable files having a name that ends with `-t`.

The following sections describe MySQL unit testing using TAP and the Google Test framework.

7.1 Unit Testing Using TAP

The unit-testing facility based on the Test Anything Protocol (TAP) is mainly used when developing Perl and PHP modules. To write unit tests for C/C++ code, MySQL has developed a library for generating TAP output from C/C++ files. Each unit test is written as a separate source file that is compiled to produce an executable. For the unit test to be recognized as a unit test, the executable file has to be of the format `mytext-t`. For example, you can create a source file named `mytest-t.c` that compiles to produce an executable `mytest-t`. The executable will be found and run when you execute `make test` or `make test-unit` in the distribution top-level directory.

Example unit tests can be found in the `unittest/examples` directory of a MySQL source distribution. The code for the MyTAP protocol is located in the `unittest/mytap` directory.

Each unit test file should be stored in a storage engine or plugin directory (`storage/`*`engine_name`* or `plugin/`*`plugin_name`*), or one of its subdirectories. A reasonable convention is to create a `unittest` subdirectory under the storage engine or plugin directory and create unit test files in `unittest`.

7.2 Unit Testing Using the Google Test Framework

The Google Test unit-testing framework is available in MySQL source trees and distributions as of MySQL 5.6.1. Google Test, like MyTAP, provides a unit-testing framework, but Google Test provides richer functionality, such as:

- A rich set of predicates
- User-defined predicates and assertions
- Automatic test registration
- Nice error reporting when a predicate fails (with line number, expected and actual values, and additional comments)
- Test fixtures, and setup/teardown logic
- Death tests
- Disabled tests
- Test filtering and shuffling

Google Test runs on many platforms. Some functionality is missing on some platforms (such as death tests and parameterized tests), so those features should not be used.

This section provides notes about using Google Test within the context of MySQL development. For general-purpose information about Google Test, see these resources:

- Main Google Test page: http://code.google.com/p/googletest
- Primer: http://code.google.com/p/googletest/wiki/GoogleTestPrimer
- Advanced guide: http://code.google.com/p/googletest/wiki/GoogleTestAdvancedGuide
- Google presentation: http://docs.google.com/present/view?id=dfsbxvm5_0f5s4pvf9

Installing Google Test and Running Unit Tests

MySQL sources do not include Google Test. To install it so that you can use it, use one of these approaches:

- **Install Google Test in individual source trees.** Use the `-DENABLE_DOWNLOADS=1` configuration option. This causes `CMake` to download Google Test and install it in your source tree for you.
- **Install a single instance of Google Test.** MySQL requires Google Test 1.6 or higher.
  - To install from a tarball, download http://googlemock.googlecode.com/files/gmock-1.6.0.zip.
  - To download Google Test from the Subversion repository, use `svn checkout http://googletest.googlecode.com/svn/trunk/ googletest-read-only`
  - If a Google Test package is available for your operating system, you can install it using the package manager. For example, you might be able to use `apt-get` for Debian Linux.

  When Google Test has been installed, set the `GTEST_PREFIX` environment variable appropriately for your command interpreter. For example, use this command line for `bash`:

```
GTEST_PREFIX=/path/to/your/install; export GTEST_PREFIX
```

Installing Google Test in individual source trees is the recommended method. The single-instance installation method can be used only if all MySQL builds on a machine take place in the same environment (same operating system, same compiler), for reasons discussed at https://groups.google.com/group/googletestframework/browse_thread/thread/668eff1cebf5309d?pli=1.

At configuration time, `CMake` looks for `gtest.h`. The build process compiles all Google Test-based unit tests or ignores them, depending on whether `gtest.h` is found.

After the build has completed, to run the tests, change location into the `unittest/gunit` directory and execute the `ctest` command. It need not be invoked with any options, but you can run it with the `--help` option to see what options are available.

Another way to run the unit tests is with this command:

```
make test-unit
```

Note

`make test-unit` is unavailable as of MySQL 5.6.32 and 5.7.14.

For internal MySQL testing, PushBuild has been extended to install Google Test as a separate “package.” All trees named `mysql-trunk.*` are set up to depend on this package.

Writing Unit Tests for Google Test

The Google Test unit test files are located in the `unittest/gunit` directory. You can look at these files to see how tests are written. Here are some examples:

- `sql_list-t.cc`: A simple test of some list classes
- `mdl-t.cc`: Some tests of metadata locking (MDL), including testing of lock acquisition from multiple threads
- `mdl_mytap-t.cc`: The same tests as `mdl-t.cc`, but written for MyTAP, to illustrate some features of Google Test

Most MyTAP-based tests are likely easily converted to Google Test. However, there might be low-level tests that absolutely must be run on every platform, and thus require MyTAP.

As currently implemented, the Google Test unit-test programs produce TAP output rather than the “plain” alternative. This can be disabled by using the `--disable-tap-output` command-line option when running a test executable.

To see what options are available, run a test executable with the `--help` option.

7.3 Unit Tests Added to Main Test Runs

From MySQL 5.5.11, `mysql-test-run.pl` will also run unit tests at the end of full test runs, when being run from within a build directory. It depends on the unit tests having been built and defined in a file `CTestTestfile.cmake` in the top level build directory. Those will normally be there after a build using `CMake`, but will not be in a binary package.

The unit tests are run simply by executing `ctest` with no arguments from the top level build directory. The result will be shown as a single test at the end, named `unit_tests` which passes if and only if all unit tests pass. A summary of the result will be printed, including the name of any failed unit tests. The set of unit tests will be counted as one test (either passed or failed) in the overall test summary.

Unit tests will by default be run only if you have not specified any specific tests or suites on the command line for `mysql-test-run.pl`. This can be overridden by setting the environment variable `MTR_UNIT_TESTS` to 0 or 1. This in turn can be overriden by a command line argument `--unit-tests` or `--nounit-tests`.

If the file `CTestTestfile.cmake` and the `ctest` command are not both available, unit tests will be silently skipped, unless you have used the command line option `--unit-tests`.

Chapter 8 Plugins for Testing Plugin Services

MySQL server plugins have access to server “services,” as described in MySQL Services for Plugins. As of MySQL 5.7.8, MySQL distributions include plugins that demonstrate how to test plugin service APIs:

- The `test_framework` plugin is a bare bones plugin that shows the minimum required framework for service testing.
- The `test_services` plugin demonstrates how to test the `my_snprintf` and `my_plugin_log_service` services in unthreaded context.
- The `test_services_threaded` plugin is like `test_services`, but for threaded context.

The source code for the plugins is located in the `plugin/test_services` directory of MySQL source distributions. The `README` file in that directory contains instructions for running the test cases available for the `test_services` and `test_services_threaded` plugins.

Note

The test plugins in `plugin/test_services` are daemon plugins (see Daemon Plugins). For an example of a nondaemon service-testing plugin plugin, see the `test_security_context.cc` file (available as of MySQL 5.7.9) in the `plugin/audit_null` directory. This file creates an audit plugin for testing the `security_context` service.

Use the following procedure to create a new service-testing plugin based on one of those provided in the `plugin/test_services` directory. Assume that you want to create a new plugin named `test_myservice` (or `test_myservice_threaded` to test in threaded context).

1. Select a source file to use as a basis for the new plugin:
   - To begin with a bare bones plugin, copy `test_framework.cc` to `test_myservice.cc`.
   - To begin with a plugin that already includes code for running tests in unthreaded context, copy `test_services.cc` to `test_myservice.cc`.
   - To begin with a plugin that already includes code for running tests in threaded context, copy `test_services_threaded.cc` to `test_myservice_threaded.cc`.
2. There is a plugin descriptor near the end of the new source file. Modify this descriptor appropriately for your plugin. Change the `name`, `author`, and `descr` members that indicate the plugin name and author and provide a description. For example, if you copied `test_framework.cc`, those members look like this:

   ```
   "test_framework",
   "Horst Hunger",
   "Test framework",
   ```

   Change them to something like this:

   ```
   "test_myservice",
   "Your Name Here",
   "Test My Service",
   ```

3. Modify your source file appropriately for the service to be tested:

- If you copied `test_framework.cc`, your file has no tests initially and is set up for unthreaded context. In this case, add code to the `test_services_plugin_init()` function. This code should invoke the service to be tested.
- If you copied `test_services.cc` or `test_services_threaded.cc`, the file contains tests for the `my_snprintf` and `my_plugin_log_service` services in unthreaded or threaded contexts. Replace or modify those tests with code for your own tests.

Compiling your plugin creates a plugin library file, which you should install in the directory named by the `plugin_dir` system variable. The file base name is the same as that of the source file. The file name suffix differs per platform (for example, `.so` for Unix and Unix-like systems, `.dll` for Windows).

To install or unintall your plugin at server startup, use the `--plugin-load` or `--plugin-load-add` option. For example, you can use these lines in an option file (adjust the file name as necessary):

```
[mysqld]
plugin-load-add=test_myservice.so
```

To install or uninstall the plugin at runtime, use these statements (adjust the plugin name and file name as necessary):

```
INSTALL PLUGIN test_myservice SONAME 'test_myservice.so';
UNINSTALL PLUGIN test_myservice;
```

For addition information about plugin loading, see Installing and Uninstalling Plugins.

For information about creating and running test cases for your new plugin, adapt the instructions in the `README` file in the `plugin/test_services` directory. Test cases for the `test_services` and `test_services_threaded` plugins are located in `mysql-test/suite/test_services`.

Index

Symbols

A

B

C

D

E

F

N

O

P

Q

R

S