MySQL 8.0 C API Developer Guide

Abstract

This is the MySQL 8.0 C API Developer Guide. This document accompanies MySQL 8.0 Reference Manual.

The C API provides low-level access to the MySQL client/server protocol and enables C programs to access database contents. The C API code is distributed with MySQL and implemented in the `libmysqlclient` library.

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

Document generated on: 2021-01-28 (revision: 68612)

Table of Contents

Preface and Legal Notices

This is the MySQL 8.0 C API Developer Guide. This document accompanies MySQL 8.0 Reference Manual.

The C API provides low-level access to the MySQL client/server protocol and enables C programs to access database contents. The C API code is distributed with MySQL and implemented in the `libmysqlclient` library.

Legal Notices

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
https://www.oracle.com/corporate/accessibility/.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit
https://www.oracle.com/corporate/accessibility/learning-support.html#support-tab.

Chapter 1 The MySQL C API

The C API provides low-level access to the MySQL client/server protocol and enables C programs to access database contents. The C API code is distributed with MySQL and implemented in the `libmysqlclient` library. See Chapter 2, *MySQL C API Implementations*.

Most other client APIs use the `libmysqlclient` library to communicate with the MySQL server. (Exceptions are Connector/J and Connector/NET.) This means that, for example, you can take advantage of many of the same environment variables that are used by other client programs because they are referenced from the library. For a list of these variables, see Overview of MySQL Programs.

For instructions on building client programs using the C API, see Section 4.1, “Building C API Client Programs”. For programming with threads, see Section 4.3, “Writing C API Threaded Client Programs”.

> **Note**
>
> If, after an upgrade, you experience problems with compiled client programs, such as `Commands out of sync` or unexpected core dumps, the programs were probably compiled using old header or library files. In this case, check the date of the `mysql.h` file and `libmysqlclient.a` library used for compilation to verify that they are from the new MySQL distribution. If not, recompile the programs with the new headers and libraries. Recompilation might also be necessary for programs compiled against the shared client library if the library major version number has changed (for example, from `libmysqlclient.so.17` to `libmysqlclient.so.18`). For additional compatibility information, see Section 4.4, “Running C API Client Programs”.

Clients have a maximum communication buffer size. The size of the buffer that is allocated initially (16KB) is automatically increased up to the maximum size (16MB by default). Because buffer sizes are increased only as demand warrants, simply increasing the maximum limit does not in itself cause more resources to be used. This size check is mostly a precaution against erroneous statements and communication packets.

The communication buffer must be large enough to contain a single SQL statement (for client-to-server traffic) and one row of returned data (for server-to-client traffic). Each session's communication buffer is dynamically enlarged to handle any query or row up to the maximum limit. For example, if you have `BLOB` values that contain up to 16MB of data, you must have a communication buffer limit of at least 16MB (in both server and client). The default maximum built into the client library is 1GB, but the default maximum in the server is 1MB. You can increase this by changing the value of the `max_allowed_packet` parameter at server startup. See Configuring the Server.

The MySQL server shrinks each communication buffer to `net_buffer_length` bytes after each query. For clients, the size of the buffer associated with a connection is not decreased until the connection is closed, at which time client memory is reclaimed.

Chapter 2 MySQL C API Implementations

The MySQL C API is a C-based API that client applications written in C can use to communicate with MySQL Server. Client programs refer to C API header files at compile time and link to a C API library file, `libmysqlclient`, at link time.

To obtain the C API header and library files required to build C API client programs, install a MySQL Server distribution.

You can install a binary distribution that contains the C API files pre-built, or you can use a source distribution and build the C API files yourself.

The names of the library files to use when linking C API client applications depend on the library type and platform for which a distribution is built:

- On Unix (and Unix-like) systems, the static library is `libmysqlclient.a`. The dynamic library is `libmysqlclient.so` on most Unix systems and `libmysqlclient.dylib` on macOS.
- On Windows, the static library is `mysqlclient.lib` and the dynamic library is `libmysql.dll`. Windows distributions also include `libmysql.lib`, a static import library needed for using the dynamic library.

  Windows distributions also include a set of debug libraries. These have the same names as the nondebug libraries, but are located in the `lib/debug` library. You must use the debug libraries when compiling clients built using the debug C runtime.

On Unix, you may also see libraries that include `_r` in the names. Before MySQL 5.5, these were built as thread-safe (re-entrant) libraries separately from the non-`_r` libraries. As of 5.5, both libraries are the same and the `_r` names are symbolic links to the corresponding non-`_r` names. There is no need to use the `_r` libraries. For example, if you use `mysql_config` to obtain linker flags, you can use `mysql_config --libs` in all cases, even for threaded clients. There is no need to use `mysql_config --libs_r`.

Chapter 3 Example C API Client Programs

Many of the clients in MySQL source distributions are written in C, such as `mysql`, `mysqladmin`, and `mysqlshow`. If you are looking for examples that demonstrate how to use the C API, take a look at those clients: Obtain a source distribution and look in its `client` directory. See How to Get MySQL.

For information about individual C API functions, the sections for most functions include usage examples.

Chapter 4 Building and Running C API Client Programs

Table of Contents

The following sections provide information on building client programs that use the C API. Topics include compiling and linking clients, writing threaded clients, and troubleshooting runtime problems.

4.1 Building C API Client Programs

This section provides guidelines for compiling C programs that use the MySQL C API.

- Compiling MySQL Clients on Unix
- Compiling MySQL Clients on Microsoft Windows
- Troubleshooting Problems Linking to the MySQL Client Library

Compiling MySQL Clients on Unix

The examples here use `gcc` as the compiler. A different compiler might be appropriate on some systems (for example, `clang` on macOS or FreeBSD, or Sun Studio on Solaris). Adjust the examples as necessary.

You may need to specify an `-I` option when you compile client programs that use MySQL header files, so that the compiler can find them. For example, if the header files are installed in `/usr/local/mysql/include`, use this option in the compile command:

```
-I/usr/local/mysql/include
```

You can link your code with either the dynamic or static MySQL C client library. The dynamic library base name is `libmysqlclient` and the suffix differs by platform (for example, `.so` for Linux, `.dylib` for macOS). The static library is named `libmysqlclient.a` on all platforms.

MySQL clients must be linked using the `-lmysqlclient` option in the link command. You may also need to specify a `-L` option to tell the linker where to find the library. For example, if the library is installed in `/usr/local/mysql/lib`, use these options in the link command:

```
-L/usr/local/mysql/lib -lmysqlclient
```

The path names may differ on your system. Adjust the `-I` and `-L` options as necessary.

To make it simpler to compile MySQL programs on Unix, use the `mysql_config` script. See `mysql_config` — Display Options for Compiling Clients.

`mysql_config` displays the options needed for compiling or linking:

```
mysql_config --cflags
mysql_config --libs
```

You can invoke those commands at the command line to get the proper options and add them manually to compilation or link commands. Alternatively, include the output from `mysql_config` directly within command lines using backticks:

```
gcc -c `mysql_config --cflags` progname.c
gcc -o progname progname.o `mysql_config --libs`
```

On Unix, linking uses dynamic libraries by default. To link to the static client library instead, add its path name to the link command. For example, if the library is located in `/usr/local/mysql/lib`, link like this:

```
gcc -o progname progname.o /usr/local/mysql/lib/libmysqlclient.a
```

Or use `mysql_config` to provide the path to the library:

```
gcc -o progname progname.o `mysql_config --variable=pkglibdir`/libmysqlclient.a
```

`mysql_config` does not currently provide a way to list all libraries needed for static linking, so it might be necessary to name additional libraries on the link command (for example, `-lnsl -lsocket` on Solaris). To get an idea which libraries to add, use `mysql_config --libs` and `ldd libmysqlclient.so` (or `otool -L libmysqlclient.dylib` on macOS).

`pkg-config` can be used as an alternative to `mysql_config` for obtaining information such as compiler flags or link libraries required to compile MySQL applications. For example, the following pairs of commands are equivalent:

```
mysql_config --cflags
pkg-config --cflags mysqlclient

mysql_config --libs
pkg-config --libs mysqlclient
```

To produce flags for static linking, use this command:

```
pkg-config --static --libs mysqlclient
```

For more information, see Section 4.2, “Building C API Client Programs Using pkg-config”.

Compiling MySQL Clients on Microsoft Windows

To specify header and library file locations, use the facilities provided by your development environment.

To build C API clients on Windows, you must link in the C client library, as well as the Windows ws2_32 sockets library and Secur32 security library.

You can link your code with either the dynamic or static MySQL C client library:

- The dynamic library is named `libmysql.dll`. In addition, the `libmysql.lib` static import library is needed for using the dynamic library.
- The static library is named `mysqlclient.lib`. To link with the static C client library, the client application must be compiled with the same version of Visual Studio used to compile the C client library (which is Visual Studio 2015 for the static C client library built by Oracle).

When using the Oracle-built MySQL C client library, follow these rules when it comes to linking the C runtime for your client application:

- For the MySQL C client library from a Community distribution of MySQL:
  - Always link dynamically to the C runtime (use the `/MD` compiler option), whether you are linking to the static or dynamic C client library. Also, target hosts running the client application must have the Visual C++ Redistributable for Visual Studio 2015 installed.
- For the MySQL C client library from a Commercial distribution of MySQL:

- If linking to the static C client library, link statically to the C runtime (use the `/MT` compiler option).
- If linking to the dynamic C client library, link either statically or dynamically to the C runtime (use either `/MT` or `/MD` compiler option).

In general, when linking to a static MySQL C client library, the client library and the client application must use the same compiler options when it comes to linking the C runtime—that is, if your C client library is compiled with the `/MT` option, your client application should also be compiled with the `/MT` option, and so on (see the MSDN page describing the C library linking options for more details). Follow this rule when you build your own static MySQL C client library from a source distribution of MySQL and link your client application to it.

> **Note**
>
> *Debug Mode:* Because of the just-mentioned linking rule, you cannot build your application in debug mode (with the `/MTd` or `/MDd` compiler option) and link it to a static C client library built by Oracle, which is *not* built with the debug options. Instead, you must build the static client library from source with the debug options.

Troubleshooting Problems Linking to the MySQL Client Library

The MySQL client library includes SSL support built in. It is unnecessary to specify either `-lssl` or `-lcrypto` at link time. Doing so may in fact result in problems at runtime.

If the linker cannot find the MySQL client library, you might get undefined-reference errors for symbols that start with `mysql_`, such as those shown here:

```
/tmp/ccFKsdPa.o: In function `main':
/tmp/ccFKsdPa.o(.text+0xb): undefined reference to `mysql_init'
/tmp/ccFKsdPa.o(.text+0x31): undefined reference to `mysql_real_connect'
/tmp/ccFKsdPa.o(.text+0x69): undefined reference to `mysql_error'
/tmp/ccFKsdPa.o(.text+0x9a): undefined reference to `mysql_close'
```

You should be able to solve this problem by adding `-L`*`dir_path`* `-lmysqlclient` at the end of your link command, where *`dir_path`* represents the path name of the directory where the client library is located. To determine the correct directory, try this command:

```
mysql_config --libs
```

The output from `mysql_config` might indicate other libraries that should be specified on the link command as well. You can include `mysql_config` output directly in your compile or link command using backticks. For example:

```
gcc -o progname progname.o `mysql_config --libs`
```

If an error occurs at link time that the `floor` symbol is undefined, link to the math library by adding `-lm` to the end of the compile/link line. Similarly, if you get undefined-reference errors for other functions that should exist on your system, such as `connect()`, check the manual page for the function in question to determine which libraries you should add to the link command.

If you get undefined-reference errors such as the following for functions that do not exist on your system, it usually means that your MySQL client library was compiled on a system that is not 100% compatible with yours:

```
mf_format.o(.text+0x201): undefined reference to `__lxstat'
```

In this case, you should download a source distribution for the latest version of MySQL and compile the MySQL client library yourself. See Installing MySQL from Source.

4.2 Building C API Client Programs Using pkg-config

MySQL distributions contain a `mysqlclient.pc` file that provides information about MySQL configuration for use by the `pkg-config` command. This enables `pkg-config` to be used as an alternative to `mysql_config` for obtaining information such as compiler flags or link libraries required to compile MySQL applications. For example, the following pairs of commands are equivalent:

```
mysql_config --cflags
pkg-config --cflags mysqlclient

mysql_config --libs
pkg-config --libs mysqlclient
```

The last `pkg-config` command produces flags for dynamic linking. To produce flags for static linking, use this command:

```
pkg-config --static --libs mysqlclient
```

On some platforms, the output with and without `--static` might be the same.

> **Note**
>
> If `pkg-config` does not find MySQL information, it might be necessary to set the `PKG_CONFIG_PATH` environment variable to the directory in which the `mysqlclient.pc` file is located, which by default is usually the `pkgconfig` directory under the MySQL library directory. For example (adjust the location appropriately):
>
> ```
> # For sh, bash, ...
> export PKG_CONFIG_PATH=/usr/local/mysql/lib/pkgconfig
> # For csh, tcsh, ...
> setenv PKG_CONFIG_PATH /usr/local/mysql/lib/pkgconfig
> ```
>
> The `mysqlconfig.pc` installation location can be controlled using the `INSTALL_PKGCONFIGDIR` `CMake` option. See MySQL Source-Configuration Options.

The `--variable` option takes a configuration variable name and displays the variable value:

```
# installation prefix directory
pkg-config --variable=prefix mysqlclient
# header file directory
pkg-config --variable=includedir mysqlclient
# library directory
pkg-config --variable=libdir mysqlclient
```

To see which variable values `pkg-config` can display using the `--variable` option, use this command:

```
pkg-config --print-variables mysqlclient
```

You can use `pkg-config` within a command line using backticks to include the output that it produces for particular options. For example, to compile and link a MySQL client program, use `pkg-config` as follows:

```
gcc -c `pkg-config --cflags mysqlclient` progname.c
gcc -o progname progname.o `pkg-config --libs mysqlclient`
```

4.3 Writing C API Threaded Client Programs

This section provides guidance for writing client programs that use the thread-related functions in the MySQL C API. For further information about these functions, see Chapter 16, *C API Threaded Function*

Descriptions. For examples of source code that uses them, look in the `client` directory of a MySQL source distribution:

- The source for `mysqlimport` uses threading in the code associated with the `--use-threads` option.
- The source for `mysqlslap` uses threads to set up simultaneous workloads, to test server operation under high load.

As an alternative to thread programming, applications may find the asynchronous (nonblocking) C API functions useful. These functions enable applications to submit multiple outstanding requests to the server and determine when each has finished using polling. For more information, see Chapter 12, *C API Asynchronous Interface*.

If undefined-reference errors occur when linking a threaded program against the MySQL client library, the most likely cause is that you did not include the thread libraries on the link/compile command.

The client library is almost thread-safe. The biggest problem is that the subroutines in `sql/net_serv.cc` that read from sockets are not interrupt-safe. This was done with the thought that you might want to have your own alarm that can break a long read to a server. If you install interrupt handlers for the `SIGPIPE` interrupt, socket handling should be thread-safe.

To avoid aborting the program when a connection terminates, MySQL blocks `SIGPIPE` on the first call to `mysql_library_init()`, `mysql_init()`, or `mysql_connect()`. To use your own `SIGPIPE` handler, first call `mysql_library_init()`, then install your handler.

The client library is thread-safe per connection. Two threads can share the same connection with the following caveats:

- Unless you are using the asynchronous C API functions mentioned previously, multiple threads cannot send a query to the MySQL server at the same time on the same connection. In particular, you must ensure that between calls to `mysql_query()` and `mysql_store_result()` in one thread, no other thread uses the same connection. To do this, use a mutex lock around your pair of `mysql_query()` and `mysql_store_result()` calls. After `mysql_store_result()` returns, the lock can be released and other threads may query the same connection.

  If you use POSIX threads, you can use `pthread_mutex_lock()` and `pthread_mutex_unlock()` to establish and release a mutex lock.

  > **Note**
  >
  > If you examine programs in a MySQL source distribution, instead of calls to `pthread_mutex_lock()` and `pthread_mutex_unlock()`, you will see calls to `native_mutex_lock()` and `native_mutex_unlock()`. The latter functions are defined in the `thr_mutex.h` header file and map to platform-specific mutex functions.

- Multiple threads can access different result sets that are retrieved with `mysql_store_result()`.
- To use `mysql_use_result()`, you must ensure that no other thread uses the same connection until the result set is closed. However, it really is best for threaded clients that share the same connection to use `mysql_store_result()`.

If a thread does not create the connection to the MySQL database but calls MySQL functions, take the following into account:

When you call `mysql_init()`, MySQL creates a thread-specific variable for the thread that is used by the debug library (among other things). If you call a MySQL function before the thread has called

mysql_init(), the thread does not have the necessary thread-specific variables in place and you are likely to end up with a core dump sooner or later. To avoid problems, you must do the following:

1. Call mysql_library_init() before any other MySQL functions. It is not thread-safe, so call it before threads are created, or protect the call with a mutex.
2. Arrange for mysql_thread_init() to be called early in the thread handler before calling any MySQL function. (If you call mysql_init(), it calls mysql_thread_init() for you.)
3. In the thread, call mysql_thread_end() before calling pthread_exit(). This frees the memory used by MySQL thread-specific variables.

The preceding notes regarding mysql_init() also apply to mysql_connect(), which calls mysql_init().

4.4 Running C API Client Programs

If, after an upgrade, you experience problems with compiled client programs, such as Commands out of sync or unexpected core dumps, the programs were probably compiled using old header or library files. In this case, check the date of the mysql.h header file and libmysqlclient.a library used for compilation to verify that they are from the new MySQL distribution. If not, recompile the programs with the new headers and libraries. Recompilation might also be necessary for programs compiled against the shared client library if the library major version number has changed (for example, from libmysqlclient.so.17 to libmysqlclient.so.18).

The major shared client library version determines compatibility. (For example, for libmysqlclient.so.18.1.0, the major version is 18.) Libraries shipped with newer versions of MySQL are drop-in replacements for older versions that have the same major number. As long as the major library version is the same, you can upgrade the library and old applications should continue to work with it.

Undefined-reference errors might occur at runtime when you try to execute a MySQL program. If these errors specify symbols that start with mysql_ or indicate that the libmysqlclient library cannot be found, it means that your system cannot find the shared libmysqlclient.so library. The solution to this problem is to tell your system to search for shared libraries in the directory where that library is located. Use whichever of the following methods is appropriate for your system:

- Add the path of the directory where libmysqlclient.so is located to the LD_LIBRARY_PATH or LD_LIBRARY environment variable.
- On macOS, add the path of the directory where libmysqlclient.dylib is located to the DYLD_LIBRARY_PATH environment variable.
- Copy the shared-library files (such as libmysqlclient.so) to some directory that is searched by your system, such as /lib, and update the shared library information by executing ldconfig. Be sure to copy all related files. A shared library might exist under several names, using symlinks to provide the alternate names.

4.5 C API Server Version and Client Library Version

The string and numeric forms of the MySQL server version are available at compile time as the values of the MYSQL_SERVER_VERSION and MYSQL_VERSION_ID macros, and at runtime as the values of the mysql_get_server_info() and mysql_get_server_version() functions.

The client library version is the MySQL version. The string and numeric forms of this version are available at compile time as the values of the MYSQL_SERVER_VERSION and MYSQL_VERSION_ID macros, and

at runtime as the values of the `mysql_get_client_info()` and `mysql_get_client_version()` functions.

Chapter 5 C API Data Structures

This section describes C API data structures other than those used for prepared statements, the asynchronous interface, or the replication stream interface. For information about those, see Chapter 9, *C API Prepared Statement Data Structures*, Chapter 13, *C API Asynchronous Interface Data Structures*, and Chapter 19, *C API Binary Log Data Structures*.

- `MYSQL`

  This structure represents the handler for one database connection. It is used for almost all MySQL functions. Do not try to make a copy of a `MYSQL` structure. There is no guarantee that such a copy will be usable.

- `MYSQL_RES`

  This structure represents the result of a query that returns rows (`SELECT`, `SHOW`, `DESCRIBE`, `EXPLAIN`). The information returned from a query is called the *result set* in the remainder of this section.

- `MYSQL_ROW`

  This is a type-safe representation of one row of data. It is currently implemented as an array of counted byte strings. (You cannot treat these as null-terminated strings if field values may contain binary data, because such values may contain null bytes internally.) Rows are obtained by calling `mysql_fetch_row()`.

- `MYSQL_FIELD`

  This structure contains metadata: information about a field, such as the field's name, type, and size. Its members are described in more detail later in this section. You may obtain the `MYSQL_FIELD` structures for each field by calling `mysql_fetch_field()` repeatedly. Field values are not part of this structure; they are contained in a `MYSQL_ROW` structure.

- `MYSQL_FIELD_OFFSET`

  This is a type-safe representation of an offset into a MySQL field list. (Used by `mysql_field_seek()`.) Offsets are field numbers within a row, beginning at zero.

- `my_ulonglong`

  A type used for 64-bit unsigned integers. The `my_ulonglong` type was used before MySQL 8.0.18. As of MySQL 8.0.18, use the `uint64_t` C type instead.

- `my_bool`

  A boolean type, for values that are true (nonzero) or false (zero). The `my_bool` type was used before MySQL 8.0. As of MySQL 8.0, use the `bool` or `int` C type instead.

  > **Note**
  >
  > The change from `my_bool` to `bool` means that the `mysql.h` header file requires a C++ or C99 compiler to compile.

The `MYSQL_FIELD` structure contains the members described in the following list. The definitions apply primarily for columns of result sets such as those produced by `SELECT` statements. `MYSQL_FIELD` structures are also used to provide metadata for `OUT` and `INOUT` parameters returned from stored procedures executed using prepared `CALL` statements. For such parameters, some of the structure members have a meaning different from the meaning for column values.

> **Tip**
>
> To interactively view the `MYSQL_FIELD` member values for result sets, invoke the `mysql` command with the `--column-type-info` option and execute some sample queries.

- `char * name`

  The name of the field, as a null-terminated string. If the field was given an alias with an `AS` clause, the value of `name` is the alias. For a procedure parameter, the parameter name.

- `char * org_name`

  The name of the field, as a null-terminated string. Aliases are ignored. For expressions, the value is an empty string. For a procedure parameter, the parameter name.

- `char * table`

  The name of the table containing this field, if it is not a calculated field. For calculated fields, the `table` value is an empty string. If the column is selected from a view, `table` names the view. If the table or view was given an alias with an `AS` clause, the value of `table` is the alias. For a `UNION`, the value is the empty string. For a procedure parameter, the procedure name.

- `char * org_table`

  The name of the table, as a null-terminated string. Aliases are ignored. If the column is selected from a view, `org_table` names the view. If the column is selected from a derived table, `org_table` names the base table. If a derived table wraps a view, `org_table` still names the base table. If the column is an expression, `org_table` is the empty string. For a `UNION`, the value is the empty string. For a procedure parameter, the value is the procedure name.

- `char * db`

  The name of the database that the field comes from, as a null-terminated string. If the field is a calculated field, `db` is an empty string. For a `UNION`, the value is the empty string. For a procedure parameter, the name of the database containing the procedure.

- `char * catalog`

  The catalog name. This value is always `"def"`.

- `char * def`

  The default value of this field, as a null-terminated string. This is set only if you use `mysql_list_fields()`.

- `unsigned long length`

  The width of the field. This corresponds to the display length, in bytes.

  The server determines the `length` value before it generates the result set, so this is the minimum length required for a data type capable of holding the largest possible value from the result column, without knowing in advance the actual values that will be produced by the query for the result set.

  For string columns, the `length` value varies on the connection character set. For example, if the character set is `latin1`, a single-byte character set, the `length` value for a `SELECT 'abc'` query is 3. If the character set is `utf8mb4`, a multibyte character set in which characters take up to 4 bytes, the `length` value is 12.

- `unsigned long max_length`

  The maximum width of the field for the result set (the length in bytes of the longest field value for the rows actually in the result set). If you use `mysql_store_result()` or `mysql_list_fields()`, this contains the maximum length for the field. If you use `mysql_use_result()`, the value of this variable is zero.

  The value of `max_length` is the length of the string representation of the values in the result set. For example, if you retrieve a `FLOAT` column and the "widest" value is `-12.345`, `max_length` is 7 (the length of `'-12.345'`).

  If you are using prepared statements, `max_length` is not set by default because for the binary protocol the lengths of the values depend on the types of the values in the result set. (See Chapter 9, *C API Prepared Statement Data Structures*.) If you want the `max_length` values anyway, enable the `STMT_ATTR_UPDATE_MAX_LENGTH` option with `mysql_stmt_attr_set()` and the lengths will be set when you call `mysql_stmt_store_result()`. (See Section 11.3, "mysql_stmt_attr_set()", and Section 11.28, "mysql_stmt_store_result()".)

- `unsigned int name_length`

  The length of `name`.

- `unsigned int org_name_length`

  The length of `org_name`.

- `unsigned int table_length`

  The length of `table`.

- `unsigned int org_table_length`

  The length of `org_table`.

- `unsigned int db_length`

  The length of `db`.

- `unsigned int catalog_length`

  The length of `catalog`.

- `unsigned int def_length`

  The length of `def`.

- `unsigned int flags`

  Bit-flags that describe the field. The `flags` value may have zero or more of the bits set that are shown in the following table.

Flag Value	Flag Description
`NOT_NULL_FLAG`	Field cannot be `NULL`
`PRI_KEY_FLAG`	Field is part of a primary key
`UNIQUE_KEY_FLAG`	Field is part of a unique key
`MULTIPLE_KEY_FLAG`	Field is part of a nonunique key

Flag Value	Flag Description
UNSIGNED_FLAG	Field has the UNSIGNED attribute
ZEROFILL_FLAG	Field has the ZEROFILL attribute
BINARY_FLAG	Field has the BINARY attribute
AUTO_INCREMENT_FLAG	Field has the AUTO_INCREMENT attribute
ENUM_FLAG	Field is an ENUM
SET_FLAG	Field is a SET
BLOB_FLAG	Field is a BLOB or TEXT (deprecated)
TIMESTAMP_FLAG	Field is a TIMESTAMP (deprecated)
NUM_FLAG	Field is numeric; see additional notes following table
NO_DEFAULT_VALUE_FLAG	Field has no default value; see additional notes following table

Some of these flags indicate data type information and are superseded by or used in conjunction with the MYSQL_TYPE_*xxx* value in the field->type member described later:

- To check for BLOB or TIMESTAMP values, check whether type is MYSQL_TYPE_BLOB or MYSQL_TYPE_TIMESTAMP. (The BLOB_FLAG and TIMESTAMP_FLAG flags are unneeded.)
- ENUM and SET values are returned as strings. For these, check that the type value is MYSQL_TYPE_STRING and that the ENUM_FLAG or SET_FLAG flag is set in the flags value.

NUM_FLAG indicates that a column is numeric. This includes columns with a type of MYSQL_TYPE_DECIMAL, MYSQL_TYPE_NEWDECIMAL, MYSQL_TYPE_TINY, MYSQL_TYPE_SHORT, MYSQL_TYPE_LONG, MYSQL_TYPE_FLOAT, MYSQL_TYPE_DOUBLE, MYSQL_TYPE_NULL, MYSQL_TYPE_LONGLONG, MYSQL_TYPE_INT24, and MYSQL_TYPE_YEAR.

NO_DEFAULT_VALUE_FLAG indicates that a column has no DEFAULT clause in its definition. This does not apply to NULL columns (because such columns have a default of NULL), or to AUTO_INCREMENT columns (which have an implied default value).

The following example illustrates a typical use of the flags value:

```
if (field->flags & NOT_NULL_FLAG)
    printf("Field cannot be null\n");
```

You may use the convenience macros shown in the following table to determine the boolean status of the flags value.

Flag Status	Description
IS_NOT_NULL(flags)	True if this field is defined as NOT NULL
IS_PRI_KEY(flags)	True if this field is a primary key
IS_BLOB(flags)	True if this field is a BLOB or TEXT (deprecated; test field->type instead)

- unsigned int decimals

  The number of decimals for numeric fields, and the fractional seconds precision for temporal fields.

- `unsigned int charsetnr`

  An ID number that indicates the character set/collation pair for the field.

  Normally, character values in result sets are converted to the character set indicated by the `character_set_results` system variable. In this case, `charsetnr` corresponds to the character set indicated by that variable. Character set conversion can be suppressed by setting `character_set_results` to `NULL`. In this case, `charsetnr` corresponds to the character set of the original table column or expression. See also Connection Character Sets and Collations.

  To distinguish between binary and nonbinary data for string data types, check whether the `charsetnr` value is 63. If so, the character set is `binary`, which indicates binary rather than nonbinary data. This enables you to distinguish `BINARY` from `CHAR`, `VARBINARY` from `VARCHAR`, and the `BLOB` types from the `TEXT` types.

  `charsetnr` values are the same as those displayed in the `Id` column of the `SHOW COLLATION` statement or the `ID` column of the `INFORMATION_SCHEMA COLLATIONS` table. You can use those information sources to see which character set and collation specific `charsetnr` values indicate:

```
mysql> SHOW COLLATION WHERE Id = 63;
+-----------+---------+----+---------+----------+---------+
| Collation | Charset | Id | Default | Compiled | Sortlen |
+-----------+---------+----+---------+----------+---------+
| binary    | binary  | 63 | Yes     | Yes      |       1 |
+-----------+---------+----+---------+----------+---------+

mysql> SELECT COLLATION_NAME, CHARACTER_SET_NAME
       FROM INFORMATION_SCHEMA.COLLATIONS WHERE ID = 33;
+-----------------+--------------------+
| COLLATION_NAME  | CHARACTER_SET_NAME |
+-----------------+--------------------+
| utf8_general_ci | utf8               |
+-----------------+--------------------+
```

- `enum enum_field_types type`

  The type of the field. The `type` value may be one of the `MYSQL_TYPE_` symbols shown in the following table.

Type Value	Type Description
`MYSQL_TYPE_TINY`	`TINYINT` field
`MYSQL_TYPE_SHORT`	`SMALLINT` field
`MYSQL_TYPE_LONG`	`INTEGER` field
`MYSQL_TYPE_INT24`	`MEDIUMINT` field
`MYSQL_TYPE_LONGLONG`	`BIGINT` field
`MYSQL_TYPE_DECIMAL`	`DECIMAL` or `NUMERIC` field
`MYSQL_TYPE_NEWDECIMAL`	Precision math `DECIMAL` or `NUMERIC`
`MYSQL_TYPE_FLOAT`	`FLOAT` field
`MYSQL_TYPE_DOUBLE`	`DOUBLE` or `REAL` field
`MYSQL_TYPE_BIT`	`BIT` field
`MYSQL_TYPE_TIMESTAMP`	`TIMESTAMP` field
`MYSQL_TYPE_DATE`	`DATE` field
`MYSQL_TYPE_TIME`	`TIME` field

Type Value	Type Description
MYSQL_TYPE_DATETIME	DATETIME field
MYSQL_TYPE_YEAR	YEAR field
MYSQL_TYPE_STRING	CHAR or BINARY field
MYSQL_TYPE_VAR_STRING	VARCHAR or VARBINARY field
MYSQL_TYPE_BLOB	BLOB or TEXT field (use max_length to determine the maximum length)
MYSQL_TYPE_SET	SET field
MYSQL_TYPE_ENUM	ENUM field
MYSQL_TYPE_GEOMETRY	Spatial field
MYSQL_TYPE_NULL	NULL-type field

The MYSQL_TYPE_TIME2, MYSQL_TYPE_DATETIME2, and MYSQL_TYPE_TIMESTAMP2) type codes are used only on the server side. Clients see the MYSQL_TYPE_TIME, MYSQL_TYPE_DATETIME, and MYSQL_TYPE_TIMESTAMP codes.

You can use the IS_NUM() macro to test whether a field has a numeric type. Pass the type value to IS_NUM() and it evaluates to TRUE if the field is numeric:

```
if (IS_NUM(field->type))
    printf("Field is numeric\n");
```

ENUM and SET values are returned as strings. For these, check that the type value is MYSQL_TYPE_STRING and that the ENUM_FLAG or SET_FLAG flag is set in the flags value.

Chapter 6 C API Function Overview

The following list summarizes the functions available in the C API. For greater detail, see the descriptions in Chapter 7, *C API Function Descriptions*.

- `mysql_affected_rows()`: Returns the number of rows changed/deleted/inserted by the last `UPDATE`, `DELETE`, or `INSERT` query.
- `mysql_autocommit()`: Toggles autocommit mode on/off.
- `mysql_bind_param()`: Defines query attributes for the next statement executed.
- `mysql_change_user()`: Changes the user and database on an open connection.
- `mysql_character_set_name()`: Returns the default character set name for the current connection.
- `mysql_client_find_plugin()`: Returns a pointer to a plugin.
- `mysql_client_register_plugin()`: Registers a plugin.
- `mysql_close()`: Closes a server connection.
- `mysql_commit()`: Commits the transaction.
- `mysql_connect()`: Connects to a MySQL server. This function is deprecated; use `mysql_real_connect()` instead.
- `mysql_create_db()`: Creates a database. This function is deprecated; use the SQL statement `CREATE DATABASE` instead.
- `mysql_data_seek()`: Seeks to an arbitrary row number in a query result set.
- `mysql_debug()`: Does a `DBUG_PUSH` with the given string.
- `mysql_drop_db()`: Drops a database. This function is deprecated; use the SQL statement `DROP DATABASE` instead.
- `mysql_dump_debug_info()`: Causes the server to write debug information to the log.
- `mysql_eof()`: Determines whether the last row of a result set has been read. This function is deprecated; use `mysql_errno()` or `mysql_error()` instead.
- `mysql_errno()`: Returns the error number for the most recently invoked MySQL function.
- `mysql_error()`: Returns the error message for the most recently invoked MySQL function.
- `mysql_escape_string()`: Escapes special characters in a string for use in an SQL statement.
- `mysql_fetch_field()`: Returns the type of the next table field.
- `mysql_fetch_field_direct()`: Returns the type of a table field, given a field number.
- `mysql_fetch_fields()`: Returns an array of all field structures.
- `mysql_fetch_lengths()`: Returns the lengths of all columns in the current row.
- `mysql_fetch_row()`: Fetches the next row from the result set.
- `mysql_field_count()`: Returns the number of result columns for the most recent statement.

- `mysql_field_seek()`: Puts the column cursor on a specified column.
- `mysql_field_tell()`: Returns the position of the field cursor used for the last `mysql_fetch_field()`.
- `mysql_free_result()`: Frees memory used by a result set.
- `mysql_get_character_set_info()`: Returns information about default character set.
- `mysql_get_client_info()`: Returns client version information as a string.
- `mysql_get_client_version()`: Returns client version information as an integer.
- `mysql_get_host_info()`: Returns a string describing the connection.
- `mysql_get_option()`: Returns the value of a `mysql_options()` option.
- `mysql_get_proto_info()`: Returns the protocol version used by the connection.
- `mysql_get_server_info()`: Returns the server version number.
- `mysql_get_server_version()`: Returns the server version number as an integer.
- `mysql_get_ssl_cipher()`: Returns the current SSL cipher.
- `mysql_hex_string()`: Encodes a string in hexadecimal format.
- `mysql_info()`: Returns information about the most recently executed query.
- `mysql_init()`: Gets or initializes a `MYSQL` structure.
- `mysql_insert_id()`: Returns the ID generated for an `AUTO_INCREMENT` column by the previous query.
- `mysql_kill()`: Kills a given thread.
- `mysql_library_end()`: Finalizes the MySQL C API library.
- `mysql_library_init()`: Initializes the MySQL C API library.
- `mysql_list_dbs()`: Returns database names matching a simple regular expression.
- `mysql_list_fields()`: Returns field names matching a simple regular expression.
- `mysql_list_processes()`: Returns a list of the current server threads.
- `mysql_list_tables()`: Returns table names matching a simple regular expression.
- `mysql_load_plugin()`: Loads a plugin.
- `mysql_load_plugin_v()`: Loads a plugin.
- `mysql_more_results()`: Checks whether any more results exist.
- `mysql_next_result()`: Returns/initiates the next result in multiple-result executions.
- `mysql_num_fields()`: Returns the number of columns in a result set.
- `mysql_num_rows()`: Returns the number of rows in a result set.

- `mysql_options()`: Sets connect options for connection-establishment functions such as `mysql_real_connect()`.
- `mysql_options4()`: Sets connect options for connection-establishment functions such as `mysql_real_connect()`.
- `mysql_ping()`: Checks whether the connection to the server is working, reconnecting as necessary.
- `mysql_plugin_options()`: Sets a plugin option.
- `mysql_query()`: Executes an SQL query specified as a null-terminated string.
- `mysql_real_connect()`: Connects to a MySQL server.
- `mysql_real_connect_dns_srv()`: Connects to a MySQL server using a DNS SRV record.
- `mysql_real_escape_string()`: Escapes special characters in a string for use in an SQL statement, taking into account the current character set of the connection.
- `mysql_real_escape_string_quote()`: Escapes special characters in a string for use in an SQL statement, taking into account the current character set of the connection and the quoting context.
- `mysql_real_query()`: Executes an SQL query specified as a counted string.
- `mysql_refresh()`: Flushes or resets tables and caches.
- `mysql_reload()`: Tells the server to reload the grant tables.
- `mysql_reset_connection()`: Resets the connection to clear session state.
- `mysql_reset_server_public_key()`: Clears a cached RSA public key from the client library.
- `mysql_result_metadata()`: Whether a result set has metadata.
- `mysql_rollback()`: Rolls back the transaction.
- `mysql_row_seek()`: Seeks to a row offset in a result set, using value returned from `mysql_row_tell()`.
- `mysql_row_tell()`: Returns the row cursor position.
- `mysql_select_db()`: Selects a database.
- `mysql_server_end()`: Finalizes the MySQL C API library.
- `mysql_server_init()`: Initializes the MySQL C API library.
- `mysql_session_track_get_first()`: Gets the first part of session state-change information.
- `mysql_session_track_get_next()`: Gets the next part of session state-change information.
- `mysql_set_character_set()`: Sets the default character set for the current connection.
- `mysql_set_local_infile_default()`: Sets the `LOAD DATA LOCAL` handler callbacks to their default values.
- `mysql_set_local_infile_handler()`: Installs application-specific `LOAD DATA LOCAL` handler callbacks.
- `mysql_set_server_option()`: Sets an option for the connection (like `multi-statements`).

- `mysql_sqlstate()`: Returns the SQLSTATE error code for the last error.
- `mysql_shutdown()`: Shuts down the database server.
- `mysql_ssl_set()`: Prepares to establish an SSL connection to the server.
- `mysql_stat()`: Returns the server status as a string.
- `mysql_store_result()`: Retrieves a complete result set to the client.
- `mysql_thread_end()`: Finalizes a thread handler.
- `mysql_thread_id()`: Returns the current thread ID.
- `mysql_thread_init()`: Initializes a thread handler.
- `mysql_thread_safe()`: Returns 1 if the clients are compiled as thread-safe.
- `mysql_use_result()`: Initiates a row-by-row result set retrieval.
- `mysql_warning_count()`: Returns the warning count for the previous SQL statement.

Application programs should use this general outline for interacting with MySQL:

1. Initialize the MySQL client library by calling `mysql_library_init()`.
2. Initialize a connection handler by calling `mysql_init()` and connect to the server by calling a connection-establishment function such as `mysql_real_connect()`.
3. Issue SQL statements and process their results. (The following discussion provides more information about how to do this.)
4. Close the connection to the MySQL server by calling `mysql_close()`.
5. End use of the MySQL client library by calling `mysql_library_end()`.

The purpose of calling `mysql_library_init()` and `mysql_library_end()` is to provide proper initialization and finalization of the MySQL client library. For applications that are linked with the client library, they provide improved memory management. If you do not call `mysql_library_end()`, a block of memory remains allocated. (This does not increase the amount of memory used by the application, but some memory leak detectors will complain about it.)

In a nonmultithreaded environment, the call to `mysql_library_init()` may be omitted, because `mysql_init()` will invoke it automatically as necessary. However, `mysql_library_init()` is not thread-safe in a multithreaded environment, and thus neither is `mysql_init()`, which calls `mysql_library_init()`. You must either call `mysql_library_init()` prior to spawning any threads, or else use a mutex to protect the call, whether you invoke `mysql_library_init()` or indirectly through `mysql_init()`. This should be done prior to any other client library call.

To connect to the server, call `mysql_init()` to initialize a connection handler, then call a connection-establishment function such as `mysql_real_connect()` with that handler (along with other information such as the host name, user name, and password). When you are done with the connection, call `mysql_close()` to terminate it. Do not use the handler after it has been closed.

Upon connection, `mysql_real_connect()` sets the `reconnect` flag (part of the `MYSQL` structure) to a value of `0`. You can use the `MYSQL_OPT_RECONNECT` option to `mysql_options()` to control reconnection behavior. Setting the flag to `1` cause the client to attempt reconnecting to the server before giving up if a statement cannot be performed because of a lost connection.

While a connection is active, the client may send SQL statements to the server using `mysql_query()` or `mysql_real_query()`. The difference between the two is that `mysql_query()` expects the query to be specified as a null-terminated string whereas `mysql_real_query()` expects a counted string. If the string contains binary data (which may include null bytes), you must use `mysql_real_query()`.

For each non-`SELECT` query (for example, `INSERT`, `UPDATE`, `DELETE`), you can find out how many rows were changed (affected) by calling `mysql_affected_rows()`.

For `SELECT` queries, you retrieve the selected rows as a result set. (Note that some statements are `SELECT`-like in that they return rows. These include `SHOW`, `DESCRIBE`, and `EXPLAIN`. Treat these statements the same way as `SELECT` statements.)

There are two ways for a client to process result sets. One way is to retrieve the entire result set all at once by calling `mysql_store_result()`. This function acquires from the server all the rows returned by the query and stores them in the client. The second way is for the client to initiate a row-by-row result set retrieval by calling `mysql_use_result()`. This function initializes the retrieval, but does not actually get any rows from the server.

In both cases, you access rows by calling `mysql_fetch_row()`. With `mysql_store_result()`, `mysql_fetch_row()` accesses rows that have previously been fetched from the server. With `mysql_use_result()`, `mysql_fetch_row()` actually retrieves the row from the server. Information about the size of the data in each row is available by calling `mysql_fetch_lengths()`.

After you are done with a result set, call `mysql_free_result()` to free the memory used for it.

The two retrieval mechanisms are complementary. Choose the approach that is most appropriate for each client application. In practice, clients tend to use `mysql_store_result()` more commonly.

An advantage of `mysql_store_result()` is that because the rows have all been fetched to the client, you not only can access rows sequentially, you can move back and forth in the result set using `mysql_data_seek()` or `mysql_row_seek()` to change the current row position within the result set. You can also find out how many rows there are by calling `mysql_num_rows()`. On the other hand, the memory requirements for `mysql_store_result()` may be very high for large result sets and you are more likely to encounter out-of-memory conditions.

An advantage of `mysql_use_result()` is that the client requires less memory for the result set because it maintains only one row at a time (and because there is less allocation overhead, `mysql_use_result()` can be faster). Disadvantages are that you must process each row quickly to avoid tying up the server, you do not have random access to rows within the result set (you can only access rows sequentially), and the number of rows in the result set is unknown until you have retrieved them all. Furthermore, you *must* retrieve all the rows even if you determine in mid-retrieval that you've found the information you were looking for.

The API makes it possible for clients to respond appropriately to statements (retrieving rows only as necessary) without knowing whether the statement is a `SELECT`. You can do this by calling `mysql_store_result()` after each `mysql_query()` (or `mysql_real_query()`). If the result set call succeeds, the statement was a `SELECT` and you can read the rows. If the result set call fails, call `mysql_field_count()` to determine whether a result was actually to be expected. If `mysql_field_count()` returns zero, the statement returned no data (indicating that it was an `INSERT`, `UPDATE`, `DELETE`, and so forth), and was not expected to return rows. If `mysql_field_count()` is nonzero, the statement should have returned rows, but did not. This indicates that the statement was a `SELECT` that failed. See the description for `mysql_field_count()` for an example of how this can be done.

Both `mysql_store_result()` and `mysql_use_result()` enable you to obtain information about the fields that make up the result set (the number of fields, their names and types, and so forth). You

can access field information sequentially within the row by calling `mysql_fetch_field()` repeatedly, or by field number within the row by calling `mysql_fetch_field_direct()`. The current field cursor position may be changed by calling `mysql_field_seek()`. Setting the field cursor affects subsequent calls to `mysql_fetch_field()`. You can also get information for fields all at once by calling `mysql_fetch_fields()`.

For detecting and reporting errors, MySQL provides access to error information by means of the `mysql_errno()` and `mysql_error()` functions. These return the error code or error message for the most recently invoked function that can succeed or fail, enabling you to determine when an error occurred and what it was.

Chapter 7 C API Function Descriptions

Table of Contents

This section describes C API functions other than those used for prepared statements, the asynchronous interface, or the replication stream interface. For information about those, see Chapter 11, *C API Prepared Statement Function Descriptions*, Chapter 15, *C API Asynchronous Function Descriptions*, and Chapter 21, *C API Binary Log Function Descriptions*.

In the descriptions here, a parameter or return value of `NULL` means `NULL` in the sense of the C programming language, not a MySQL `NULL` value.

Functions that return a value generally return a pointer or an integer. Unless specified otherwise, functions returning a pointer return a non-`NULL` value to indicate success or a `NULL` value to indicate an error, and functions returning an integer return zero to indicate success or nonzero to indicate an error. Note that “nonzero” means just that. Unless the function description says otherwise, do not test against a value other than zero:

```
if (result)                  /* correct */
    ... error ...

if (result < 0)              /* incorrect */
    ... error ...
```

```
if (result == -1)             /* incorrect */
    ... error ...
```

When a function returns an error, the **Errors** subsection of the function description lists the possible types of errors. You can find out which of these occurred by calling `mysql_errno()`. A string representation of the error may be obtained by calling `mysql_error()`.

7.1 mysql_affected_rows()

```
uint64_t
mysql_affected_rows(MYSQL *mysql)
```

Description

`mysql_affected_rows()` may be called immediately after executing a statement with `mysql_query()` or `mysql_real_query()`. It returns the number of rows changed, deleted, or inserted by the last statement if it was an `UPDATE`, `DELETE`, or `INSERT`. For `SELECT` statements, `mysql_affected_rows()` works like `mysql_num_rows()`.

For `UPDATE` statements, the affected-rows value by default is the number of rows actually changed. If you specify the `CLIENT_FOUND_ROWS` flag to `mysql_real_connect()` when connecting to `mysqld`, the affected-rows value is the number of rows “found”; that is, matched by the `WHERE` clause.

For `REPLACE` statements, the affected-rows value is 2 if the new row replaced an old row, because in this case, one row was inserted after the duplicate was deleted.

For `INSERT ... ON DUPLICATE KEY UPDATE` statements, the affected-rows value per row is 1 if the row is inserted as a new row, 2 if an existing row is updated, and 0 if an existing row is set to its current values. If you specify the `CLIENT_FOUND_ROWS` flag, the affected-rows value is 1 (not 0) if an existing row is set to its current values.

Following a `CALL` statement for a stored procedure, `mysql_affected_rows()` returns the value that it would return for the last statement executed within the procedure, or `0` if that statement would return `-1`. Within the procedure, you can use `ROW_COUNT()` at the SQL level to obtain the affected-rows value for individual statements.

`mysql_affected_rows()` returns a meaningful value for a wide range of statements. For details, see the description for `ROW_COUNT()` in Information Functions.

Return Values

An integer greater than zero indicates the number of rows affected or retrieved. Zero indicates that no records were updated for an `UPDATE` statement, no rows matched the `WHERE` clause in the query or that no query has yet been executed. -1 indicates that the query returned an error or that, for a `SELECT` query, `mysql_affected_rows()` was called prior to calling `mysql_store_result()`.

Because `mysql_affected_rows()` returns an unsigned value, you can check for -1 by comparing the return value to `(uint64_t)-1` (or to `(uint64_t)~0`, which is equivalent).

Errors

None.

Example

```
char *stmt = "UPDATE products SET cost=cost*1.25
```

```
              WHERE group=10";
mysql_query(&mysql,stmt);
printf("%ld products updated",
       (long) mysql_affected_rows(&mysql));
```

7.2 mysql_autocommit()

```
bool
mysql_autocommit(MYSQL *mysql,
                 bool mode)
```

Description

Sets autocommit mode on if `mode` is 1, off if `mode` is 0.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

None.

7.3 mysql_bind_param()

```
bool
mysql_bind_param(MYSQL *mysql,
                 unsigned n_params,
                 MYSQL_BIND *bind,
                 const char **name)
```

Description

`mysql_bind_param()`, available as of MySQL 8.0.23, enables defining attributes that apply to the next query sent to the server. For discussion of the purpose and use of query attributes, see Query Attributes.

Attributes defined with `mysql_bind_param()` apply to nonprepared statements executed in blocking fashion with `mysql_query()` or `mysql_real_query()`, or in nonblocking fashion with `mysql_real_query_nonblocking()`. Attributes do not apply to prepared statements executed with `mysql_stmt_execute()`.

If multiple `mysql_bind_param()` calls occur prior to query execution, only the last call applies.

Attributes defined with `mysql_bind_param()` apply only to the next query executed and are cleared thereafter. The `mysql_reset_connection()` and `mysql_change_user()` functions also clear any currently defined attributes.

`mysql_bind_param()` is backward compatible. For connections to older servers that do not support query attributes, no attributes are sent.

Arguments:

- `mysql`: The connection handler returned from `mysql_init()`.
- `n_params`: The number of attributes defined by the `bind` and `name` arguments.

- `bind`: The address of an array of `MYSQL_BIND` structures. The array should contain `n_params` elements, one for each attribute.
- `name`: The address of an array of character pointers, each pointing to a null-terminated string defining an attribute name. The array should contain `n_params` elements, one for each attribute. Query attribute names are transmitted using the character set indicated by the `character_set_client` system variable.

Each attribute has a name, a value, and a data type. The `name` argument defines attribute names, and the `bind` argument defines their values and types. For a description of the members of the `MYSQL_BIND` data structure used for the `bind` argument, see Chapter 9, *C API Prepared Statement Data Structures*.

Each attribute type most be one of the `MYSQL_TYPE_xxx` types listed in Table 9.1, "Permissible Input Data Types for MYSQL_BIND Structures", except that `MYSQL_TYPE_BLOB` and `MYSQL_TYPE_TEXT` are not supported. If an unsupported type is specified for an attribute, a `CR_UNSUPPORTED_PARAM_TYPE` error occurs.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_UNSUPPORTED_PARAM_TYPE`

  The attribute data type is not supported.

Example

This example uses `mysql_bind_param()` to define string and integer query attributes, then retrieves and displays their values by name using the `mysql_query_attribute_string()` user-defined function:

```
MYSQL_BIND bind[2];
const char *name[2] = { "name1", "name2" };
char *char_data = "char value";
int int_data = 3;
unsigned long length[2] = { 10, sizeof(int) };
int status;

/* clear and initialize attribute butffers */
memset(bind, 0, sizeof (bind));

bind[0].buffer_type = MYSQL_TYPE_STRING;
bind[0].buffer = char_data;
bind[0].length = &length[0];
bind[0].is_null = 0;

bind[1].buffer_type = MYSQL_TYPE_LONG;
bind[1].buffer = (char *) &int_data;
bind[1].length = &length[1];
bind[1].is_null = 0;

/* bind attributes */
status = mysql_bind_param(&mysql, 2, bind, name);
test_error(&mysql, status);
const char *query =
"SELECT mysql_query_attribute_string('name1'),"
"       mysql_query_attribute_string('name2')";
status = mysql_real_query(&mysql, query, strlen(query));
test_error(&mysql, status);
```

```
MYSQL_RES *result = mysql_store_result(&mysql);
MYSQL_ROW row = mysql_fetch_row(result);
unsigned long *lengths = mysql_fetch_lengths(result);
for(int i = 0; i < 2; i++)
{
    printf("attribute %d: [%.*s]\n", i+1, (int) lengths[i],
           row[i] ? row[i] : "NULL");
}
mysql_free_result(result);
```

When executed, the code produces this result:

```
attribute 1: [char value]
attribute 2: [3]
```

7.4 mysql_change_user()

```
bool
mysql_change_user(MYSQL *mysql,
                  const char *user,
                  const char *password,
                  const char *db)
```

Description

Changes the user and causes the database specified by `db` to become the default (current) database on the connection specified by `mysql`. In subsequent queries, this database is the default for table references that include no explicit database specifier.

`mysql_change_user()` fails if the connected user cannot be authenticated or does not have permission to use the database. In this case, the user and database are not changed.

Pass a `db` parameter of `NULL` if you do not want to have a default database.

This function resets the session state as if one had done a new connect and reauthenticated. (See Chapter 28, *C API Automatic Reconnection Control*.) It always performs a `ROLLBACK` of any active transactions, closes and drops all temporary tables, and unlocks all locked tables. It resets session system variables to the values of the corresponding global system variables, releases prepared statements, closes `HANDLER` variables, and releases locks acquired with `GET_LOCK()`. Clears any current query attributes defined as a result of calling `mysql_bind_param()`. These effects occur even if the user did not change.

To reset the connection state in a more lightweight manner without changing the user, use `mysql_reset_connection()`.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

The same that you can get from `mysql_real_connect()`, plus:

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- CR_SERVER_LOST

  The connection to the server was lost during the query.

- CR_UNKNOWN_ERROR

  An unknown error occurred.

- ER_UNKNOWN_COM_ERROR

  The MySQL server does not implement this command (probably an old server).

- ER_ACCESS_DENIED_ERROR

  The user or password was wrong.

- ER_BAD_DB_ERROR

  The database did not exist.

- ER_DBACCESS_DENIED_ERROR

  The user did not have access rights to the database.

- ER_WRONG_DB_NAME

  The database name was too long.

Example

```
if (mysql_change_user(&mysql, "user", "password", "new_database"))
{
   fprintf(stderr, "Failed to change user.  Error: %s\n",
           mysql_error(&mysql));
}
```

7.5 mysql_character_set_name()

```
const char *
mysql_character_set_name(MYSQL *mysql)
```

Description

Returns the default character set name for the current connection.

Return Values

The default character set name

Errors

None.

7.6 mysql_close()

```
void
mysql_close(MYSQL *mysql)
```

Description

Closes a previously opened connection. `mysql_close()` also deallocates the connection handler pointed to by `mysql` if the handler was allocated automatically by `mysql_init()` or `mysql_connect()`. Do not use the handler after it has been closed.

Return Values

None.

Errors

None.

7.7 mysql_commit()

```
bool
mysql_commit(MYSQL *mysql)
```

Description

Commits the current transaction.

The action of this function is subject to the value of the `completion_type` system variable. In particular, if the value of `completion_type` is `RELEASE` (or 2), the server performs a release after terminating a transaction and closes the client connection. Call `mysql_close()` from the client program to close the connection from the client side.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

None.

7.8 mysql_connect()

```
MYSQL *
mysql_connect(MYSQL *mysql,
              const char *host,
              const char *user,
              const char *passwd)
```

Description

This function is deprecated. Use `mysql_real_connect()` instead.

7.9 mysql_create_db()

```
int
mysql_create_db(MYSQL *mysql,
                const char *db)
```

Description

Creates the database named by the `db` parameter.

This function is deprecated. Use `mysql_query()` to issue an SQL `CREATE DATABASE` statement instead.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

Example

```
if(mysql_create_db(&mysql, "my_database"))
{
   fprintf(stderr, "Failed to create new database.  Error: %s\n",
           mysql_error(&mysql));
}
```

7.10 mysql_data_seek()

```
void
mysql_data_seek(MYSQL_RES *result,
                uint64_t offset)
```

Description

Seeks to an arbitrary row in a query result set. The `offset` value is a row number. Specify a value in the range from `0` to `mysql_num_rows(result)-1`.

This function requires that the result set structure contains the entire result of the query, so `mysql_data_seek()` may be used only in conjunction with `mysql_store_result()`, not with `mysql_use_result()`.

Return Values

None.

Errors

None.

7.11 mysql_debug()

```
void
mysql_debug(const char *debug)
```

Description

Does a DBUG_PUSH with the given string. mysql_debug() uses the Fred Fish debug library. To use this function, you must compile the client library to support debugging. See The DBUG Package.

Return Values

None.

Errors

None.

Example

The call shown here causes the client library to generate a trace file in /tmp/client.trace on the client machine:

```
mysql_debug("d:t:O,/tmp/client.trace");
```

7.12 mysql_drop_db()

```
int
mysql_drop_db(MYSQL *mysql,
              const char *db)
```

Description

Drops the database named by the db parameter.

This function is deprecated. Use mysql_query() to issue an SQL DROP DATABASE statement instead.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- CR_COMMANDS_OUT_OF_SYNC

  Commands were executed in an improper order.

- CR_SERVER_GONE_ERROR

  The MySQL server has gone away.

- CR_SERVER_LOST

  The connection to the server was lost during the query.

- CR_UNKNOWN_ERROR

An unknown error occurred.

Example

```
if(mysql_drop_db(&mysql, "my_database"))
  fprintf(stderr, "Failed to drop the database: Error: %s\n",
          mysql_error(&mysql));
```

7.13 mysql_dump_debug_info()

```
int
mysql_dump_debug_info(MYSQL *mysql)
```

Description

Instructs the server to write debugging information to the error log. The connected user must have the `SUPER` privilege.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

7.14 mysql_eof()

```
bool
mysql_eof(MYSQL_RES *result)
```

Description

This function is deprecated. `mysql_errno()` or `mysql_error()` may be used instead.

`mysql_eof()` determines whether the last row of a result set has been read.

If you acquire a result set from a successful call to `mysql_store_result()`, the client receives the entire set in one operation. In this case, a `NULL` return from `mysql_fetch_row()` always means the end of the result set has been reached and it is unnecessary to call `mysql_eof()`. When used with `mysql_store_result()`, `mysql_eof()` always returns true.

On the other hand, if you use `mysql_use_result()` to initiate a result set retrieval, the rows of the set are obtained from the server one by one as you call `mysql_fetch_row()` repeatedly. Because an error may occur on the connection during this process, a `NULL` return value from `mysql_fetch_row()` does not necessarily mean the end of the result set was reached normally. In this case, you can use `mysql_eof()` to determine what happened. `mysql_eof()` returns a nonzero value if the end of the result set was reached and zero if an error occurred.

Historically, `mysql_eof()` predates the standard MySQL error functions `mysql_errno()` and `mysql_error()`. Because those error functions provide the same information, their use is preferred over `mysql_eof()`, which is deprecated. (In fact, they provide more information, because `mysql_eof()` returns only a boolean value whereas the error functions indicate a reason for the error when one occurs.)

Return Values

Zero for success. Nonzero if the end of the result set has been reached.

Errors

None.

Example

The following example shows how you might use `mysql_eof()`:

```
mysql_query(&mysql,"SELECT * FROM some_table");
result = mysql_use_result(&mysql);
while((row = mysql_fetch_row(result)))
{
    // do something with data
}
if(!mysql_eof(result))  // mysql_fetch_row() failed due to an error
{
    fprintf(stderr, "Error: %s\n", mysql_error(&mysql));
}
```

However, you can achieve the same effect with the standard MySQL error functions:

```
mysql_query(&mysql,"SELECT * FROM some_table");
result = mysql_use_result(&mysql);
while((row = mysql_fetch_row(result)))
{
    // do something with data
}
if(mysql_errno(&mysql))  // mysql_fetch_row() failed due to an error
{
    fprintf(stderr, "Error: %s\n", mysql_error(&mysql));
}
```

7.15 mysql_errno()

```
unsigned int
mysql_errno(MYSQL *mysql)
```

Description

For the connection specified by `mysql`, `mysql_errno()` returns the error code for the most recently invoked API function that can succeed or fail. A return value of zero means that no error occurred. Client error message numbers are listed in the MySQL `errmsg.h` header file. Server error message numbers are listed in `mysqld_error.h`. Errors also are listed at Error Messages and Common Problems.

> **Note**
>
> Some functions such as `mysql_fetch_row()` do not set `mysql_errno()` if they succeed. A rule of thumb is that all functions that have to ask the server for information reset `mysql_errno()` if they succeed.

MySQL-specific error numbers returned by `mysql_errno()` differ from SQLSTATE values returned by `mysql_sqlstate()`. For example, the `mysql` client program displays errors using the following format, where `1146` is the `mysql_errno()` value and `'42S02'` is the corresponding `mysql_sqlstate()` value:

```
shell> SELECT * FROM no_such_table;
ERROR 1146 (42S02): Table 'test.no_such_table' doesn't exist
```

Return Values

An error code value for the last `mysql_xxx()` call, if it failed. zero means no error occurred.

Errors

None.

7.16 mysql_error()

```
const char *
mysql_error(MYSQL *mysql)
```

Description

For the connection specified by `mysql`, `mysql_error()` returns a null-terminated string containing the error message for the most recently invoked API function that failed. If a function did not fail, the return value of `mysql_error()` may be the previous error or an empty string to indicate no error.

A rule of thumb is that all functions that have to ask the server for information reset `mysql_error()` if they succeed.

For functions that reset `mysql_error()`, either of these two tests can be used to check for an error:

```
if(*mysql_error(&mysql))
{
  // an error occurred
}

if(mysql_error(&mysql)[0])
{
  // an error occurred
}
```

The language of the client error messages may be changed by recompiling the MySQL client library. You can choose error messages in several different languages. See Setting the Error Message Language.

Return Values

A null-terminated character string that describes the error. An empty string if no error occurred.

Errors

None.

7.17 mysql_escape_string()

> **Note**
>
> Do not use this function. `mysql_escape_string()` does not have arguments that enable it to respect the current character set or the quoting context. Use `mysql_real_escape_string_quote()` instead.

7.18 mysql_fetch_field()

```
MYSQL_FIELD *
mysql_fetch_field(MYSQL_RES *result)
```

Description

Returns the definition of one column of a result set as a `MYSQL_FIELD` structure. Call this function repeatedly to retrieve information about all columns in the result set. `mysql_fetch_field()` returns `NULL` when no more fields are left.

For metadata-optional connections, this function returns `NULL` when the `resultset_metadata` system variable is set to `NONE`. To check whether a result set has metadata, use the `mysql_result_metadata()` function. For details about managing result set metadata transfer, see Chapter 27, *C API Optional Result Set Metadata*.

`mysql_fetch_field()` is reset to return information about the first field each time you execute a new `SELECT` query. The field returned by `mysql_fetch_field()` is also affected by calls to `mysql_field_seek()`.

If you've called `mysql_query()` to perform a `SELECT` on a table but have not called `mysql_store_result()`, MySQL returns the default blob length (8KB) if you call `mysql_fetch_field()` to ask for the length of a `BLOB` field. (The 8KB size is chosen because MySQL does not know the maximum length for the `BLOB`. This should be made configurable sometime.) Once you've retrieved the result set, `field->max_length` contains the length of the largest value for this column in the specific query.

Return Values

The `MYSQL_FIELD` structure for the current column. `NULL` if no columns are left or the result set has no metadata.

Errors

None.

Example

```
MYSQL_FIELD *field;

while((field = mysql_fetch_field(result)))
{
    printf("field name %s\n", field->name);
}
```

7.19 mysql_fetch_field_direct()

```
MYSQL_FIELD *
```

```
mysql_fetch_field_direct(MYSQL_RES *result,
                         unsigned int fieldnr)
```

Description

Given a field number `fieldnr` for a column within a result set, returns that column's field definition as a `MYSQL_FIELD` structure. Use this function to retrieve the definition for an arbitrary column. Specify a value for `fieldnr` in the range from 0 to `mysql_num_fields(result)-1`.

For metadata-optional connections, this function returns `NULL` when the `resultset_metadata` system variable is set to `NONE`. To check whether a result set has metadata, use the `mysql_result_metadata()` function. For details about managing result set metadata transfer, see Chapter 27, *C API Optional Result Set Metadata*.

Return Values

The `MYSQL_FIELD` structure for the specified column. `NULL` if the result set has no metadata.

Errors

None.

Example

```
unsigned int num_fields;
unsigned int i;
MYSQL_FIELD *field;

num_fields = mysql_num_fields(result);
for(i = 0; i < num_fields; i++)
{
    field = mysql_fetch_field_direct(result, i);
    printf("Field %u is %s\n", i, field->name);
}
```

7.20 mysql_fetch_fields()

```
MYSQL_FIELD *
mysql_fetch_fields(MYSQL_RES *result)
```

Description

Returns an array of all `MYSQL_FIELD` structures for a result set. Each structure provides the field definition for one column of the result set.

For metadata-optional connections, this function returns `NULL` when the `resultset_metadata` system variable is set to `NONE`. To check whether a result set has metadata, use the `mysql_result_metadata()` function. For details about managing result set metadata transfer, see Chapter 27, *C API Optional Result Set Metadata*.

Return Values

An array of `MYSQL_FIELD` structures for all columns of a result set. `NULL` if the result set has no metadata.

Errors

None.

Example

```
unsigned int num_fields;
unsigned int i;
MYSQL_FIELD *fields;

num_fields = mysql_num_fields(result);
fields = mysql_fetch_fields(result);
for(i = 0; i < num_fields; i++)
{
   printf("Field %u is %s\n", i, fields[i].name);
}
```

7.21 mysql_fetch_lengths()

```
unsigned long *
mysql_fetch_lengths(MYSQL_RES *result)
```

Description

Returns the lengths of the columns of the current row within a result set. If you plan to copy field values, this length information is also useful for optimization, because you can avoid calling `strlen()`. In addition, if the result set contains binary data, you **must** use this function to determine the size of the data, because `strlen()` returns incorrect results for any field containing null characters.

The length for empty columns and for columns containing `NULL` values is zero. To see how to distinguish these two cases, see the description for `mysql_fetch_row()`.

Return Values

An array of unsigned long integers representing the size of each column (not including any terminating null bytes). `NULL` if an error occurred.

Errors

`mysql_fetch_lengths()` is valid only for the current row of the result set. It returns `NULL` if you call it before calling `mysql_fetch_row()` or after retrieving all rows in the result.

Example

```
MYSQL_ROW row;
unsigned long *lengths;
unsigned int num_fields;
unsigned int i;

row = mysql_fetch_row(result);
if (row)
{
    num_fields = mysql_num_fields(result);
    lengths = mysql_fetch_lengths(result);
    for(i = 0; i < num_fields; i++)
    {
         printf("Column %u is %lu bytes in length.\n",
                i, lengths[i]);
    }
}
```

7.22 mysql_fetch_row()

```
MYSQL_ROW
mysql_fetch_row(MYSQL_RES *result)
```

Description

> **Note**
>
> `mysql_fetch_row()` is a synchronous function. Its asynchronous counterpart is `mysql_fetch_row_nonblocking()`, for use by applications that require asynchronous communication with the server. See Chapter 12, *C API Asynchronous Interface*.

`mysql_fetch_row()` retrieves the next row of a result set:

- When used after `mysql_store_result()` or `mysql_store_result_nonblocking()`, `mysql_fetch_row()` returns `NULL` if there are no more rows to retrieve.
- When used after `mysql_use_result()`, `mysql_fetch_row()` returns `NULL` if there are no more rows to retrieve or an error occurred.

The number of values in the row is given by `mysql_num_fields(result)`. If `row` holds the return value from a call to `mysql_fetch_row()`, pointers to the values are accessed as `row[0]` to `row[mysql_num_fields(result)-1]`. `NULL` values in the row are indicated by `NULL` pointers.

The lengths of the field values in the row may be obtained by calling `mysql_fetch_lengths()`. Empty fields and fields containing `NULL` both have length 0; you can distinguish these by checking the pointer for the field value. If the pointer is `NULL`, the field is `NULL`; otherwise, the field is empty.

Return Values

A `MYSQL_ROW` structure for the next row, or `NULL`. The meaning of a `NULL` return depends on which function was called preceding `mysql_fetch_row()`:

- When used after `mysql_store_result()` or `mysql_store_result_nonblocking()`, `mysql_fetch_row()` returns `NULL` if there are no more rows to retrieve.
- When used after `mysql_use_result()`, `mysql_fetch_row()` returns `NULL` if there are no more rows to retrieve or an error occurred. To determine whether an error occurred, check whether `mysql_error()` returns a nonempty string or `mysql_errno()` returns nonzero.

Errors

Errors are not reset between calls to `mysql_fetch_row()`

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

Example

```
MYSQL_ROW row;
unsigned int num_fields;
unsigned int i;
```

```
num_fields = mysql_num_fields(result);
while ((row = mysql_fetch_row(result)))
{
   unsigned long *lengths;
   lengths = mysql_fetch_lengths(result);
   for(i = 0; i < num_fields; i++)
   {
       printf("[%.*s] ", (int) lengths[i],
              row[i] ? row[i] : "NULL");
   }
   printf("\n");
}
```

7.23 mysql_field_count()

```
unsigned int
mysql_field_count(MYSQL *mysql)
```

Description

Returns the number of columns for the most recent query on the connection.

The normal use of this function is when `mysql_store_result()` returned `NULL` (and thus you have no result set pointer). In this case, you can call `mysql_field_count()` to determine whether `mysql_store_result()` should have produced a nonempty result. This enables the client program to take proper action without knowing whether the query was a `SELECT` (or `SELECT`-like) statement. The example shown here illustrates how this may be done.

See Section 29.1, “Why mysql_store_result() Sometimes Returns NULL After mysql_query() Returns Success”.

Return Values

An unsigned integer representing the number of columns in a result set.

Errors

None.

Example

```
MYSQL_RES *result;
unsigned int num_fields;
unsigned int num_rows;

if (mysql_query(&mysql,query_string))
{
    // error
}
else // query succeeded, process any data returned by it
{
    result = mysql_store_result(&mysql);
    if (result)  // there are rows
    {
        num_fields = mysql_num_fields(result);
        // retrieve rows, then call mysql_free_result(result)
    }
    else  // mysql_store_result() returned nothing; should it have?
    {
        if(mysql_field_count(&mysql) == 0)
        {
```

```
            // query does not return data
            // (it was not a SELECT)
            num_rows = mysql_affected_rows(&mysql);
        }
        else // mysql_store_result() should have returned data
        {
            fprintf(stderr, "Error: %s\n", mysql_error(&mysql));
        }
    }
}
```

An alternative is to replace the `mysql_field_count(&mysql)` call with `mysql_errno(&mysql)`. In this case, you are checking directly for an error from `mysql_store_result()` rather than inferring from the value of `mysql_field_count()` whether the statement was a `SELECT`.

7.24 mysql_field_seek()

```
MYSQL_FIELD_OFFSET
mysql_field_seek(MYSQL_RES *result,
                 MYSQL_FIELD_OFFSET offset)
```

Description

Sets the field cursor to the given offset. The next call to `mysql_fetch_field()` retrieves the field definition of the column associated with that offset.

To seek to the beginning of a row, pass an `offset` value of zero.

Return Values

The previous value of the field cursor.

Errors

None.

7.25 mysql_field_tell()

```
MYSQL_FIELD_OFFSET
mysql_field_tell(MYSQL_RES *result)
```

Description

Returns the position of the field cursor used for the last `mysql_fetch_field()`. This value can be used as an argument to `mysql_field_seek()`.

Return Values

The current offset of the field cursor.

Errors

None.

7.26 mysql_free_result()

```
void
```

```
mysql_free_result(MYSQL_RES *result)
```

Description

> **Note**
>
> `mysql_free_result()` is a synchronous function. Its asynchronous counterpart is `mysql_free_result_nonblocking()`, for use by applications that require asynchronous communication with the server. See Chapter 12, *C API Asynchronous Interface*.

`mysql_free_result()` frees the memory allocated for a result set by `mysql_store_result()`, `mysql_use_result()`, `mysql_list_dbs()`, and so forth. When you are done with a result set, you must free the memory it uses by calling `mysql_free_result()`.

Do not attempt to access a result set after freeing it.

Return Values

None.

Errors

None.

7.27 mysql_get_character_set_info()

```
void
mysql_get_character_set_info(MYSQL *mysql,
                             MY_CHARSET_INFO *cs)
```

Description

This function provides information about the default client character set. The default character set may be changed with the `mysql_set_character_set()` function.

Example

This example shows the fields that are available in the `MY_CHARSET_INFO` structure:

```
if (!mysql_set_character_set(&mysql, "utf8"))
{
    MY_CHARSET_INFO cs;
    mysql_get_character_set_info(&mysql, &cs);
    printf("character set information:\n");
    printf("character set+collation number: %d\n", cs.number);
    printf("character set name: %s\n", cs.name);
    printf("collation name: %s\n", cs.csname);
    printf("comment: %s\n", cs.comment);
    printf("directory: %s\n", cs.dir);
    printf("multi byte character min. length: %d\n", cs.mbminlen);
    printf("multi byte character max. length: %d\n", cs.mbmaxlen);
}
```

7.28 mysql_get_client_info()

```
const char *
mysql_get_client_info(void)
```

Description

Returns a string that represents the MySQL client library version (for example, `"8.0.25"`).

The function value is the version of MySQL that provides the client library. For more information, see Section 4.5, “C API Server Version and Client Library Version”.

Return Values

A character string that represents the MySQL client library version.

Errors

None.

7.29 mysql_get_client_version()

```
unsigned long
mysql_get_client_version(void)
```

Description

Returns an integer that represents the MySQL client library version. The value has the format *XYYZZ* where *X* is the major version, *YY* is the release level (or minor version), and *ZZ* is the sub-version within the release level:

```
major_version*10000 + release_level*100 + sub_version
```

For example, `"8.0.25"` is returned as `80025`.

The function value is the version of MySQL that provides the client library. For more information, see Section 4.5, “C API Server Version and Client Library Version”.

Return Values

An integer that represents the MySQL client library version.

Errors

None.

7.30 mysql_get_host_info()

```
const char *
mysql_get_host_info(MYSQL *mysql)
```

Description

Returns a string describing the type of connection in use, including the server host name.

Return Values

A character string representing the server host name and the connection type.

Errors

None.

7.31 mysql_get_option()

```
int
mysql_get_option(MYSQL *mysql,
                 enum mysql_option option,
                 const void *arg)
```

Description

Returns the current value of an option settable using `mysql_options()`. The value should be treated as read only.

The `option` argument is the option for which you want its value. The `arg` argument is a pointer to a variable in which to store the option value. `arg` must be a pointer to a variable of the type appropriate for the `option` argument. The following table shows which variable type to use for each `option` value.

For `MYSQL_OPT_MAX_ALLOWED_PACKET`, it is possible to set a session or global maximum buffer size, depending on whether the `mysql` argument to `mysql_options()` is non-`NULL` or `NULL`, `mysql_get_option()` similarly returns the session or global value depending on its `mysql` argument.

`arg` Type	**Applicable `option` Values**
`unsigned int`	`MYSQL_OPT_CONNECT_TIMEOUT`, `MYSQL_OPT_PROTOCOL`, `MYSQL_OPT_READ_TIMEOUT`, `MYSQL_OPT_RETRY_COUNT`, `MYSQL_OPT_SSL_FIPS_MODE`, `MYSQL_OPT_SSL_MODE`, `MYSQL_OPT_WRITE_TIMEOUT`, `MYSQL_OPT_ZSTD_COMPRESSION_LEVEL`
`unsigned long`	`MYSQL_OPT_MAX_ALLOWED_PACKET`, `MYSQL_OPT_NET_BUFFER_LENGTH`
`bool`	`MYSQL_ENABLE_CLEARTEXT_PLUGIN`, `MYSQL_OPT_CAN_HANDLE_EXPIRED_PASSWORDS`, `MYSQL_OPT_GET_SERVER_PUBLIC_KEY`, `MYSQL_OPT_LOCAL_INFILE`, `MYSQL_OPT_OPTIONAL_RESULTSET_METADATA`, `MYSQL_OPT_RECONNECT`, `MYSQL_REPORT_DATA_TRUNCATION`
`const char *`	`MYSQL_DEFAULT_AUTH`, `MYSQL_OPT_BIND`, `MYSQL_OPT_COMPRESSION_ALGORITHMS`, `MYSQL_OPT_LOAD_DATA_LOCAL_DIR`, `MYSQL_OPT_SSL_CA`, `MYSQL_OPT_SSL_CAPATH`, `MYSQL_OPT_SSL_CERT`, `MYSQL_OPT_SSL_CIPHER`, `MYSQL_OPT_SSL_CRL`, `MYSQL_OPT_SSL_CRLPATH`, `MYSQL_OPT_SSL_KEY`, `MYSQL_OPT_TLS_CIPHERSUITES`, `MYSQL_OPT_TLS_VERSION`, `MYSQL_PLUGIN_DIR`, `MYSQL_READ_DEFAULT_FILE`, `MYSQL_READ_DEFAULT_GROUP`, `MYSQL_SERVER_PUBLIC_KEY`, `MYSQL_SET_CHARSET_DIR`, `MYSQL_SET_CHARSET_NAME`, `MYSQL_SHARED_MEMORY_BASE_NAME`
argument not used	`MYSQL_OPT_COMPRESS`
cannot be queried (error is returned)	`MYSQL_INIT_COMMAND`, `MYSQL_OPT_CONNECT_ATTR_DELETE`, `MYSQL_OPT_CONNECT_ATTR_RESET`, `MYSQL_OPT_NAMED_PIPE`

Return Values

Zero for success. Nonzero if an error occurred; this occurs for `option` values that cannot be queried.

Example

The following call tests the `MYSQL_OPT_RECONNECT` option. After the call returns successfully, the value of `reconnect` is true or false to indicate whether automatic reconnection is enabled.

```
bool reconnect;

if (mysql_get_option(mysql, MYSQL_OPT_RECONNECT, &reconnect))
  fprintf(stderr, "mysql_get_option() failed\n");
```

7.32 mysql_get_proto_info()

```
unsigned int
mysql_get_proto_info(MYSQL *mysql)
```

Description

Returns the protocol version used by current connection.

Return Values

An unsigned integer representing the protocol version used by the current connection.

Errors

None.

7.33 mysql_get_server_info()

```
const char *
mysql_get_server_info(MYSQL *mysql)
```

Description

Returns a string that represents the MySQL server version (for example, `"8.0.25"`).

Return Values

A character string that represents the MySQL server version.

Errors

None.

7.34 mysql_get_server_version()

```
unsigned long
mysql_get_server_version(MYSQL *mysql)
```

Description

Returns an integer that represents the MySQL server version. The value has the format *XYYZZ* where *X* is the major version, *YY* is the release level (or minor version), and *ZZ* is the sub-version within the release level:

```
major_version*10000 + release_level*100 + sub_version
```

For example, `"8.0.25"` is returned as `80025`.

This function is useful in client programs for determining whether some version-specific server capability exists.

Return Values

An integer that represents the MySQL server version.

Errors

None.

7.35 mysql_get_ssl_cipher()

```
const char *
mysql_get_ssl_cipher(MYSQL *mysql)
```

Description

`mysql_get_ssl_cipher()` returns the encryption cipher used for the given connection to the server. `mysql` is the connection handler returned from `mysql_init()`.

Return Values

A string naming the encryption cipher used for the connection, or `NULL` if the connection is not encrypted.

7.36 mysql_hex_string()

```
unsigned long
mysql_hex_string(char *to,
                 const char *from,
                 unsigned long length)
```

Description

This function creates a legal SQL string for use in an SQL statement. See String Literals.

The string in the `from` argument is encoded in hexadecimal format, with each character encoded as two hexadecimal digits. The result is placed in the `to` argument, followed by a terminating null byte.

The string pointed to by `from` must be `length` bytes long. You must allocate the `to` buffer to be at least `length*2+1` bytes long. When `mysql_hex_string()` returns, the contents of `to` is a null-terminated string. The return value is the length of the encoded string, not including the terminating null byte.

The return value can be placed into an SQL statement using either `X'value'` or `0xvalue` format. However, the return value does not include the `X'...'` or `0x`. The caller must supply whichever of those is desired.

Example

```
char query[1000],*end;

end = strmov(query,"INSERT INTO test_table values(");
end = strmov(end,"X'");
end += mysql_hex_string(end,"What is this",12);
end = strmov(end,"',X'");
end += mysql_hex_string(end,"binary data: \0\r\n",16);
end = strmov(end,"')");

if (mysql_real_query(&mysql,query,(unsigned int) (end - query)))
{
   fprintf(stderr, "Failed to insert row, Error: %s\n",
```

```
            mysql_error(&mysql));
}
```

The `strmov()` function used in the example is included in the `libmysqlclient` library and works like `strcpy()` but returns a pointer to the terminating null of the first parameter.

Return Values

The length of the encoded string that is placed into `to`, not including the terminating null character.

Errors

None.

7.37 mysql_info()

```
const char *
mysql_info(MYSQL *mysql)
```

Description

Retrieves a string providing information about the most recently executed statement, but only for the statements listed here. For other statements, `mysql_info()` returns `NULL`. The format of the string varies depending on the type of statement, as described here. The numbers are illustrative only; the string contains values appropriate for the statement.

- `INSERT INTO ... SELECT ...`

  String format: `Records: 100 Duplicates: 0 Warnings: 0`

- `INSERT INTO ... VALUES (...),(...),(...)...`

  String format: `Records: 3 Duplicates: 0 Warnings: 0`

- `LOAD DATA`

  String format: `Records: 1 Deleted: 0 Skipped: 0 Warnings: 0`

- `ALTER TABLE`

  String format: `Records: 3 Duplicates: 0 Warnings: 0`

- `UPDATE`

  String format: `Rows matched: 40 Changed: 40 Warnings: 0`

`mysql_info()` returns a non-`NULL` value for `INSERT ... VALUES` only for the multiple-row form of the statement (that is, only if multiple value lists are specified).

Return Values

A character string representing additional information about the most recently executed statement. `NULL` if no information is available for the statement.

Errors

None.

7.38 mysql_init()

```
MYSQL *
mysql_init(MYSQL *mysql)
```

Description

Allocates or initializes a `MYSQL` object suitable for `mysql_real_connect()`. If `mysql` is a `NULL` pointer, the function allocates, initializes, and returns a new object. Otherwise, the object is initialized and the address of the object is returned. If `mysql_init()` allocates a new object, it is freed when `mysql_close()` is called to close the connection.

In a nonmultithreaded environment, `mysql_init()` invokes `mysql_library_init()` automatically as necessary. However, `mysql_library_init()` is not thread-safe in a multithreaded environment, and thus neither is `mysql_init()`. Before calling `mysql_init()`, either call `mysql_library_init()` prior to spawning any threads, or use a mutex to protect the `mysql_library_init()` call. This should be done prior to any other client library call.

Return Values

An initialized `MYSQL*` handler. `NULL` if there was insufficient memory to allocate a new object.

Errors

In case of insufficient memory, `NULL` is returned.

7.39 mysql_insert_id()

```
uint64_t
mysql_insert_id(MYSQL *mysql)
```

Description

Returns the value generated for an `AUTO_INCREMENT` column by the previous `INSERT` or `UPDATE` statement. Use this function after you have performed an `INSERT` statement into a table that contains an `AUTO_INCREMENT` field, or have used `INSERT` or `UPDATE` to set a column value with `LAST_INSERT_ID(expr)`.

The return value of `mysql_insert_id()` is always zero unless explicitly updated under one of the following conditions:

- `INSERT` statements that store a value into an `AUTO_INCREMENT` column. This is true whether the value is automatically generated by storing the special values `NULL` or `0` into the column, or is an explicit nonspecial value.
- In the case of a multiple-row `INSERT` statement, `mysql_insert_id()` returns the first automatically generated `AUTO_INCREMENT` value that was successfully inserted.

  If no rows are successfully inserted, `mysql_insert_id()` returns 0.
- If an `INSERT ... SELECT` statement is executed, and no automatically generated value is successfully inserted, `mysql_insert_id()` returns the ID of the last inserted row.
- If an `INSERT ... SELECT` statement uses `LAST_INSERT_ID(expr)`, `mysql_insert_id()` returns *expr*.

- `INSERT` statements that generate an `AUTO_INCREMENT` value by inserting `LAST_INSERT_ID(expr)` into any column or by updating any column to `LAST_INSERT_ID(expr)`.
- If the previous statement returned an error, the value of `mysql_insert_id()` is undefined.

The return value of `mysql_insert_id()` can be simplified to the following sequence:

1. If there is an `AUTO_INCREMENT` column, and an automatically generated value was successfully inserted, return the first such value.
2. If `LAST_INSERT_ID(expr)` occurred in the statement, return `expr`, even if there was an `AUTO_INCREMENT` column in the affected table.
3. The return value varies depending on the statement used. When called after an `INSERT` statement:
   - If there is an `AUTO_INCREMENT` column in the table, and there were some explicit values for this column that were successfully inserted into the table, return the last of the explicit values.

   When called after an `INSERT ... ON DUPLICATE KEY UPDATE` statement:

   - If there is an `AUTO_INCREMENT` column in the table and there were some explicit successfully inserted values or some updated values, return the last of the inserted or updated values.

`mysql_insert_id()` returns `0` if the previous statement does not use an `AUTO_INCREMENT` value. If you must save the value for later, be sure to call `mysql_insert_id()` immediately after the statement that generates the value.

The value of `mysql_insert_id()` is affected only by statements issued within the current client connection. It is not affected by statements issued by other clients.

The `LAST_INSERT_ID()` SQL function will contain the value of the first automatically generated value that was successfully inserted. `LAST_INSERT_ID()` is not reset between statements because the value of that function is maintained in the server. Another difference from `mysql_insert_id()` is that `LAST_INSERT_ID()` is not updated if you set an `AUTO_INCREMENT` column to a specific nonspecial value. See Information Functions.

`mysql_insert_id()` returns `0` following a `CALL` statement for a stored procedure that generates an `AUTO_INCREMENT` value because in this case `mysql_insert_id()` applies to `CALL` and not the statement within the procedure. Within the procedure, you can use `LAST_INSERT_ID()` at the SQL level to obtain the `AUTO_INCREMENT` value.

The reason for the differences between `LAST_INSERT_ID()` and `mysql_insert_id()` is that `LAST_INSERT_ID()` is made easy to use in scripts while `mysql_insert_id()` tries to provide more exact information about what happens to the `AUTO_INCREMENT` column.

> **Note**
>
> The OK packet used in the client/server protocol holds information such as is used for session state tracking. When clients read the OK packet to know whether there is a session state change, this resets values such as the last insert ID and the number of affected rows. Such changes cause `mysql_insert_id()` to return 0 after execution of commands including but not necessarily limited to `COM_PING`, `COM_REFRESH`, and `COM_INIT_DB`.

Return Values

Described in the preceding discussion.

Errors

- ER_AUTO_INCREMENT_CONFLICT

  A user-specified AUTO_INCREMENT value in a multi INSERT statement falls within the range between the current AUTO_INCREMENT value and the sum of the current and number of rows affected values.

7.40 mysql_kill()

```
int
mysql_kill(MYSQL *mysql,
           unsigned long pid)
```

Description

> **Note**
>
> mysql_kill() is deprecated and will be removed in a future version of MySQL. Instead, use mysql_query() to execute a KILL statement.

Asks the server to kill the thread specified by pid.

This function is deprecated. Use mysql_query() to issue an SQL KILL statement instead.

mysql_kill() cannot handle values larger than 32 bits, but to guard against killing the wrong thread returns an error in these cases:

- If given an ID larger than 32 bits, mysql_kill() returns a CR_INVALID_CONN_HANDLE error.
- After the server's internal thread ID counter reaches a value larger than 32 bits, it returns an ER_DATA_OUT_OF_RANGE error for any mysql_kill() invocation and mysql_kill() fails.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- CR_COMMANDS_OUT_OF_SYNC

  Commands were executed in an improper order.

- CR_INVALID_CONN_HANDLE

  The pid was larger than 32 bits.

- CR_SERVER_GONE_ERROR

  The MySQL server has gone away.

- CR_SERVER_LOST

  The connection to the server was lost during the query.

- CR_UNKNOWN_ERROR

  An unknown error occurred.

- ER_DATA_OUT_OF_RANGE

  The server's internal thread ID counter has reached a value larger than 32 bits, at which point it rejects all mysql_kill() invocations.

7.41 mysql_library_end()

```
void
mysql_library_end(void)
```

Description

This function finalizes the MySQL client library. Call it when you are done using the library (for example, after disconnecting from the server).

> **Note**
>
> To avoid memory leaks after the application is done using the library (for example, after closing the connection to the server), be sure to call mysql_library_end() explicitly. This enables memory managment to be performed to clean up and free resources used by the library.

For usage information, see Chapter 6, *C API Function Overview*, and Section 7.42, “mysql_library_init()”.

7.42 mysql_library_init()

```
int
mysql_library_init(int argc,
                   char **argv,
                   char **groups)
```

Description

Call this function to initialize the MySQL client library before you call any other MySQL function.

> **Note**
>
> To avoid memory leaks after the application is done using the library (for example, after closing the connection to the server), be sure to call mysql_library_end() explicitly. This enables memory managment to be performed to clean up and free resources used by the library. See Section 7.41, “mysql_library_end()”.

In a nonmultithreaded environment, the call to mysql_library_init() may be omitted, because mysql_init() will invoke it automatically as necessary. However, mysql_library_init() is not thread-safe in a multithreaded environment, and thus neither is mysql_init(), which calls mysql_library_init(). You must either call mysql_library_init() prior to spawning any threads, or else use a mutex to protect the call, whether you invoke mysql_library_init() or indirectly through mysql_init(). Do this prior to any other client library call.

The argc, argv, and groups arguments are unused. In older MySQL versions, they were used for applications linked against the embedded server, which is no longer supported. The call now should be written as mysql_library_init(0, NULL, NULL).

```
#include <mysql.h>
#include <stdlib.h>

int main(void) {
  if (mysql_library_init(0, NULL, NULL)) {
```

```
    fprintf(stderr, "could not initialize MySQL client library\n");
    exit(1);
  }

  /* Use any MySQL API functions here */

  mysql_library_end();

  return EXIT_SUCCESS;
}
```

Return Values

Zero for success. Nonzero if an error occurred.

7.43 mysql_list_dbs()

```
MYSQL_RES *
mysql_list_dbs(MYSQL *mysql,
               const char *wild)
```

Description

Returns a result set consisting of database names on the server that match the simple regular expression specified by the `wild` parameter. `wild` may contain the wildcard characters `%` or `_`, or may be a `NULL` pointer to match all databases. Calling `mysql_list_dbs()` is similar to executing the query `SHOW DATABASES [LIKE wild]`.

You must free the result set with `mysql_free_result()`.

Return Values

A `MYSQL_RES` result set for success. `NULL` if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_OUT_OF_MEMORY`

  Out of memory.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

7.44 mysql_list_fields()

```
MYSQL_RES *
```

```
mysql_list_fields(MYSQL *mysql,
                  const char *table,
                  const char *wild)
```

Description

> **Note**
>
> `mysql_list_fields()` is deprecated and will be removed in a future version of MySQL. Instead, use `mysql_query()` to execute a `SHOW COLUMNS` statement.

Returns an empty result set for which the metadata provides information about the columns in the given table that match the simple regular expression specified by the `wild` parameter. `wild` may contain the wildcard characters `%` or `_`, or may be a `NULL` pointer to match all fields. Calling `mysql_list_fields()` is similar to executing the query `SHOW COLUMNS FROM tbl_name [LIKE wild]`.

It is preferable to use `SHOW COLUMNS FROM tbl_name` instead of `mysql_list_fields()`.

You must free the result set with `mysql_free_result()`.

Return Values

A `MYSQL_RES` result set for success. `NULL` if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

Example

```
int i;
MYSQL_RES *tbl_cols = mysql_list_fields(mysql, "mytbl", "f%");

unsigned int field_cnt = mysql_num_fields(tbl_cols);
printf("Number of columns: %d\n", field_cnt);

for (i=0; i < field_cnt; ++i)
{
  /* col describes i-th column of the table */
  MYSQL_FIELD *col = mysql_fetch_field_direct(tbl_cols, i);
  printf ("Column %d: %s\n", i, col->name);
}
mysql_free_result(tbl_cols);
```

7.45 mysql_list_processes()

```
MYSQL_RES *
mysql_list_processes(MYSQL *mysql)
```

Description

> **Note**
>
> `mysql_list_processes()` is deprecated and will be removed in a future version of MySQL. Instead, use `mysql_query()` to execute a `SHOW PROCESSLIST` statement.

Returns a result set describing the current server threads. This is the same kind of information as that reported by `mysqladmin processlist` or a `SHOW PROCESSLIST` query.

You must free the result set with `mysql_free_result()`.

Return Values

A `MYSQL_RES` result set for success. `NULL` if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

7.46 mysql_list_tables()

```
MYSQL_RES *
mysql_list_tables(MYSQL *mysql,
                  const char *wild)
```

Description

Returns a result set consisting of table names in the current database that match the simple regular expression specified by the `wild` parameter. `wild` may contain the wildcard characters `%` or `_`, or may be a `NULL` pointer to match all tables. Calling `mysql_list_tables()` is similar to executing the query `SHOW TABLES [LIKE wild]`.

You must free the result set with `mysql_free_result()`.

Return Values

A `MYSQL_RES` result set for success. `NULL` if an error occurred.

Errors

- CR_COMMANDS_OUT_OF_SYNC

  Commands were executed in an improper order.

- CR_SERVER_GONE_ERROR

  The MySQL server has gone away.

- CR_SERVER_LOST

  The connection to the server was lost during the query.

- CR_UNKNOWN_ERROR

  An unknown error occurred.

7.47 mysql_more_results()

```
bool
mysql_more_results(MYSQL *mysql)
```

Description

This function is used when you execute multiple statements specified as a single statement string, or when you execute CALL statements, which can return multiple result sets.

mysql_more_results() true if more results exist from the currently executed statement, in which case the application must call mysql_next_result() to fetch the results.

Return Values

TRUE (1) if more results exist. FALSE (0) if no more results exist.

In most cases, you can call mysql_next_result() instead to test whether more results exist and initiate retrieval if so.

See Chapter 23, *C API Multiple Statement Execution Support*, and Section 7.48, “mysql_next_result()”.

Errors

None.

7.48 mysql_next_result()

```
int
mysql_next_result(MYSQL *mysql)
```

Description

> **Note**
>
> mysql_next_result() is a synchronous function. Its asynchronous counterpart is mysql_next_result_nonblocking(), for use by applications that require asynchronous communication with the server. See Chapter 12, *C API Asynchronous Interface*.

`mysql_next_result()` is used when you execute multiple statements specified as a single statement string, or when you use `CALL` statements to execute stored procedures, which can return multiple result sets.

`mysql_next_result()` reads the next statement result and returns a status to indicate whether more results exist. If `mysql_next_result()` returns an error, there are no more results.

Before each call to `mysql_next_result()`, you must call `mysql_free_result()` for the current statement if it is a statement that returned a result set (rather than just a result status).

After calling `mysql_next_result()` the state of the connection is as if you had called `mysql_real_query()` or `mysql_query()` for the next statement. This means that you can call `mysql_store_result()`, `mysql_warning_count()`, `mysql_affected_rows()`, and so forth.

If your program uses `CALL` statements to execute stored procedures, the `CLIENT_MULTI_RESULTS` flag must be enabled. This is because each `CALL` returns a result to indicate the call status, in addition to any result sets that might be returned by statements executed within the procedure. Because `CALL` can return multiple results, process them using a loop that calls `mysql_next_result()` to determine whether there are more results.

`CLIENT_MULTI_RESULTS` can be enabled when you call `mysql_real_connect()`, either explicitly by passing the `CLIENT_MULTI_RESULTS` flag itself, or implicitly by passing `CLIENT_MULTI_STATEMENTS` (which also enables `CLIENT_MULTI_RESULTS`). `CLIENT_MULTI_RESULTS` is enabled by default.

It is also possible to test whether there are more results by calling `mysql_more_results()`. However, this function does not change the connection state, so if it returns true, you must still call `mysql_next_result()` to advance to the next result.

For an example that shows how to use `mysql_next_result()`, see Chapter 23, *C API Multiple Statement Execution Support*.

Return Values

Return Value	Description
0	Successful and there are more results
-1	Successful and there are no more results
>0	An error occurred

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order. For example, if you did not call `mysql_use_result()` for a previous result set.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

7.49 mysql_num_fields()

```
unsigned int
mysql_num_fields(MYSQL_RES *result)
```

To pass a `MYSQL*` argument instead, use `unsigned int mysql_field_count(MYSQL *mysql)`.

Description

Returns the number of columns in a result set.

You can get the number of columns either from a pointer to a result set or to a connection handler. You would use the connection handler if `mysql_store_result()` or `mysql_use_result()` returned `NULL` (and thus you have no result set pointer). In this case, you can call `mysql_field_count()` to determine whether `mysql_store_result()` should have produced a nonempty result. This enables the client program to take proper action without knowing whether the query was a `SELECT` (or `SELECT`-like) statement. The example shown here illustrates how this may be done.

See Section 29.1, “Why mysql_store_result() Sometimes Returns NULL After mysql_query() Returns Success”.

Return Values

An unsigned integer representing the number of columns in a result set.

Errors

None.

Example

```
MYSQL_RES *result;
unsigned int num_fields;
unsigned int num_rows;

if (mysql_query(&mysql,query_string))
{
    // error
}
else // query succeeded, process any data returned by it
{
    result = mysql_store_result(&mysql);
    if (result)  // there are rows
    {
        num_fields = mysql_num_fields(result);
        // retrieve rows, then call mysql_free_result(result)
    }
    else  // mysql_store_result() returned nothing; should it have?
    {
        if (mysql_errno(&mysql))
        {
           fprintf(stderr, "Error: %s\n", mysql_error(&mysql));
        }
        else if (mysql_field_count(&mysql) == 0)
        {
            // query does not return data
            // (it was not a SELECT)
            num_rows = mysql_affected_rows(&mysql);
        }
    }
}
```

An alternative (if you know that your query should have returned a result set) is to replace the `mysql_errno(&mysql)` call with a check whether `mysql_field_count(&mysql)` returns 0. This happens only if something went wrong.

7.50 mysql_num_rows()

```
uint64_t
mysql_num_rows(MYSQL_RES *result)
```

Description

Returns the number of rows in the result set.

The use of `mysql_num_rows()` depends on whether you use `mysql_store_result()` or `mysql_use_result()` to return the result set. If you use `mysql_store_result()`, `mysql_num_rows()` may be called immediately. If you use `mysql_use_result()`, `mysql_num_rows()` does not return the correct value until all the rows in the result set have been retrieved.

`mysql_num_rows()` is intended for use with statements that return a result set, such as `SELECT`. For statements such as `INSERT`, `UPDATE`, or `DELETE`, the number of affected rows can be obtained with `mysql_affected_rows()`.

Return Values

The number of rows in the result set.

Errors

None.

7.51 mysql_options()

```
int
mysql_options(MYSQL *mysql,
              enum mysql_option option,
              const void *arg)
```

Description

Can be used to set extra connect options and affect behavior for a connection. This function may be called multiple times to set several options. To retrieve option values, use `mysql_get_option()`.

Call `mysql_options()` after `mysql_init()` and before `mysql_connect()` or `mysql_real_connect()`.

The `option` argument is the option that you want to set; the `arg` argument is the value for the option. If the option is an integer, specify a pointer to the value of the integer as the `arg` argument.

Options for information such as SSL certificate and key files are used to establish an encrypted connection if such connections are available, but do not enforce any requirement that the connection obtained be encrypted. To require an encrypted connection, use the technique described in Chapter 22, *C API Support for Encrypted Connections*.

The following list describes the possible options, their effect, and how `arg` is used for each option. For option descriptions that indicate `arg` is unused, its value is irrelevant; it is conventional to pass 0.

- `MYSQL_DEFAULT_AUTH` (argument type: `char *`)

  The name of the authentication plugin to use.

- `MYSQL_ENABLE_CLEARTEXT_PLUGIN` (argument type: `bool *`)

  Enable the `mysql_clear_password` cleartext authentication plugin. See Client-Side Cleartext Pluggable Authentication.

- `MYSQL_INIT_COMMAND` (argument type: `char *`)

  SQL statement to execute when connecting to the MySQL server. Automatically re-executed if reconnection occurs.

- `MYSQL_OPT_BIND` (argument: `char *`)

  The network interface from which to connect to the server. This is used when the client host has multiple network interfaces. The argument is a host name or IP address (specified as a string).

- `MYSQL_OPT_CAN_HANDLE_EXPIRED_PASSWORDS` (argument type: `bool *`)

  Indicate whether the client can handle expired passwords. See Server Handling of Expired Passwords.

- `MYSQL_OPT_COMPRESS` (argument: not used)

  Compress all information sent between the client and the server if possible. See Connection Compression Control.

  As of MySQL 8.0.18, `MYSQL_OPT_COMPRESS` becomes a legacy option, due to the introduction of the `MYSQL_OPT_COMPRESSION_ALGORITHMS` option for more control over connection compression (see Configuring Connection Compression). The meaning of `MYSQL_OPT_COMPRESS` depends on whether `MYSQL_OPT_COMPRESSION_ALGORITHMS` is specified:

  - When `MYSQL_OPT_COMPRESSION_ALGORITHMS` is not specified, enabling `MYSQL_OPT_COMPRESS` is equivalent to specifying a client-side algorithm set of `zlib,uncompressed`.

  - When `MYSQL_OPT_COMPRESSION_ALGORITHMS` is specified, enabling `MYSQL_OPT_COMPRESS` is equivalent to specifying an algorithm set of `zlib` and the full client-side algorithm set is the union of `zlib` plus the algorithms specified by `MYSQL_OPT_COMPRESSION_ALGORITHMS`. For example, with `MYSQL_OPT_COMPRESS` enabled and `MYSQL_OPT_COMPRESSION_ALGORITHMS` set to `zlib,zstd`, the permitted-algorithm set is `zlib` plus `zlib,zstd`; that is, `zlib,zstd`. With `MYSQL_OPT_COMPRESS` enabled and `MYSQL_OPT_COMPRESSION_ALGORITHMS` set to `zstd,uncompressed`, the permitted-algorithm set is `zlib` plus `zstd,uncompressed`; that is, `zlib,zstd,uncompressed`.

  As of MySQL 8.0.18, `MYSQL_OPT_COMPRESS` is deprecated. It will be removed in a future MySQL version. See Configuring Legacy Connection Compression.

- `MYSQL_OPT_COMPRESSION_ALGORITHMS` (argument type: `const char *`)

  The permitted compression algorithms for connections to the server. The available algorithms are the same as for the `protocol_compression_algorithms` system variable. If this option is not specified, the default value is `uncompressed`.

  For more information, see Connection Compression Control.

  This option was added in MySQL 8.0.18. For asynchronous operations, the option has no effect until MySQL 8.0.21.

- MYSQL_OPT_CONNECT_ATTR_DELETE (argument type: char *)

  Given a key name, this option deletes a key-value pair from the current set of connection attributes to pass to the server at connect time. The argument is a pointer to a null-terminated string naming the key. Comparison of the key name with existing keys is case-sensitive.

  See also the description for the MYSQL_OPT_CONNECT_ATTR_RESET option, as well as the description for the MYSQL_OPT_CONNECT_ATTR_ADD option in the description of the mysql_options4() function. That function description also includes a usage example.

  The Performance Schema exposes connection attributes through the session_connect_attrs and session_account_connect_attrs tables. See Performance Schema Connection Attribute Tables.

- MYSQL_OPT_CONNECT_ATTR_RESET (argument not used)

  This option resets (clears) the current set of connection attributes to pass to the server at connect time.

  See also the description for the MYSQL_OPT_CONNECT_ATTR_DELETE option, as well as the description for the MYSQL_OPT_CONNECT_ATTR_ADD option in the description of the mysql_options4() function. That function description also includes a usage example.

  The Performance Schema exposes connection attributes through the session_connect_attrs and session_account_connect_attrs tables. See Performance Schema Connection Attribute Tables.

- MYSQL_OPT_CONNECT_TIMEOUT (argument type: unsigned int *)

  The connect timeout in seconds.

- MYSQL_OPT_GET_SERVER_PUBLIC_KEY (argument type: bool *)

  Enables the client to request from the server the public key required for RSA key pair-based password exchange. This option applies to clients that authenticate with the caching_sha2_password authentication plugin. For that plugin, the server does not send the public key unless requested. This option is ignored for accounts that do not authenticate with that plugin. It is also ignored if RSA-based password exchange is not used, as is the case when the client connects to the server using a secure connection.

  If MYSQL_SERVER_PUBLIC_KEY is given and specifies a valid public key file, it takes precedence over MYSQL_OPT_GET_SERVER_PUBLIC_KEY.

  For information about the caching_sha2_password plugin, see Caching SHA-2 Pluggable Authentication.

- MYSQL_OPT_LOAD_DATA_LOCAL_DIR (argument type: char *)

  This option affects the client-side LOCAL capability for LOAD DATA operations. It specifies the directory in which files named in LOAD DATA LOCAL statements must be located. The effect of MYSQL_OPT_LOAD_DATA_LOCAL_DIR depends on whether LOCAL data loading is enabled or disabled:

  - If LOCAL data loading is enabled, either by default in the MySQL client library or by explicitly enabling MYSQL_OPT_LOCAL_INFILE, the MYSQL_OPT_LOAD_DATA_LOCAL_DIR option has no effect.

  - If LOCAL data loading is disabled, either by default in the MySQL client library or by explicitly disabling MYSQL_OPT_LOCAL_INFILE, the MYSQL_OPT_LOAD_DATA_LOCAL_DIR option can be used to designate a permitted directory for locally loaded files. In this case, LOCAL data loading is permitted but restricted to files located in the designated directory. Interpretation of the MYSQL_OPT_LOAD_DATA_LOCAL_DIR value is as follows:

- If the value is the null pointer (the default), it names no directory, with the result that no files are permitted for `LOCAL` data loading.
- If the value is a directory path name, `LOCAL` data loading is permitted but restricted to files located in the named directory. Comparison of the directory path name and the path name of files to be loaded is case-sensitive regardless of the case-sensitivity of the underlying file system.

For example, to explicitly disable local data loading except for files located in the `/my/local/data` directory, invoke `mysql_options()` like this:

```
unsigned int i = 0;
mysql_options(&mysql,MYSQL_OPT_LOCAL_INFILE,&i);
mysql_options(&mysql,MYSQL_OPT_LOAD_DATA_LOCAL_DIR,"/my/local/data");
```

The `MYSQL_OPT_LOAD_DATA_LOCAL_DIR` option can be set any time during the life of the `mysql` connection handler. Once set, the value applies to all subsequent `LOCAL` load operations until such time as the value is changed.

The `ENABLED_LOCAL_INFILE` `CMake` option controls the client library default for local data loading (see MySQL Source-Configuration Options).

Successful use of `LOCAL` load operations by a client also requires that the server permits local loading; see Security Considerations for LOAD DATA LOCAL

The `MYSQL_OPT_LOAD_DATA_LOCAL_DIR` option was added in MySQL 8.0.21.

- `MYSQL_OPT_LOCAL_INFILE` (argument type: optional pointer to `unsigned int`)

  This option affects client-side `LOCAL` capability for `LOAD DATA` operations. By default, `LOCAL` capability is determined by the default compiled into the MySQL client library. To control this capability explicitly, invoke `mysql_options()` to enable or disable the `MYSQL_OPT_LOCAL_INFILE` option:

  - To enable `LOCAL` data loading, set the pointer to point to an `unsigned int` that has a nonzero value, or omit the pointer argument.
  - To disable `LOCAL` data loading, set the pointer to point to an `unsigned int` that has a zero value.

  If `LOCAL` capability is disabled, the `MYSQL_OPT_LOAD_DATA_LOCAL_DIR` option can be used to permit restricted local loading of files located in a designated directory.

  The `ENABLED_LOCAL_INFILE` `CMake` option controls the client library default for local data loading (see MySQL Source-Configuration Options).

  Successful use of `LOCAL` load operations by a client also requires that the server permits local loading; see Security Considerations for LOAD DATA LOCAL

- `MYSQL_OPT_MAX_ALLOWED_PACKET` (argument: `unsigned long *`)

  This option sets the client-side maximum size of the buffer for client/server communication. If the `mysql` argument is non-`NULL`, the call sets the option value for that session. If `mysql` is `NULL`, the call sets the option value globally for all subsequent sessions for which a session-specific value is not specified.

  Because it is possible to set a session or global maximum buffer size, depending on whether the `mysql` argument is non-`NULL` or `NULL`, `mysql_get_option()` similarly returns the session or global value depending on its `mysql` argument.

- `MYSQL_OPT_NAMED_PIPE` (argument: not used)

Use a named pipe to connect to the MySQL server on Windows, if the server permits named-pipe connections.

- `MYSQL_OPT_NET_BUFFER_LENGTH` (argument: `unsigned long *`)

  This option sets the client-side buffer size for TCP/IP and socket communication.

- `MYSQL_OPT_OPTIONAL_RESULTSET_METADATA` (argument type: `bool *`)

  This flag makes result set metadata optional. It is an alternative to setting the `CLIENT_OPTIONAL_RESULTSET_METADATA` connection flag for the `mysql_real_connect()` function. For details about managing result set metadata transfer, see Chapter 27, *C API Optional Result Set Metadata*.

- `MYSQL_OPT_PROTOCOL` (argument type: `unsigned int *`)

  Transport protocol to use for connection. Specify one of the enum values of `mysql_protocol_type` defined in `mysql.h`.

- `MYSQL_OPT_READ_TIMEOUT` (argument type: `unsigned int *`)

  The timeout in seconds for each attempt to read from the server. There are retries if necessary, so the total effective timeout value is three times the option value. You can set the value so that a lost connection can be detected earlier than the TCP/IP `Close_Wait_Timeout` value of 10 minutes.

- `MYSQL_OPT_RECONNECT` (argument type: `bool *`)

  Enable or disable automatic reconnection to the server if the connection is found to have been lost. Reconnect is off by default; this option provides a way to set reconnection behavior explicitly. See Chapter 28, *C API Automatic Reconnection Control*.

- `MYSQL_OPT_RETRY_COUNT` (argument type: `unsigned int *`)

  The retry count for I/O-related system calls that are interrupted while connecting to the server or communicating with it. If this option is not specified, the default value is 1 (1 retry if the initial call is interrupted for 2 tries total).

  This option can be used only by clients that link against a C client library compiled with NDB Cluster support.

- `MYSQL_OPT_SSL_CA` (argument type: `char *`)

  The path name of the Certificate Authority (CA) certificate file. This option, if used, must specify the same certificate used by the server.

- `MYSQL_OPT_SSL_CAPATH` (argument type: `char *`)

  The path name of the directory that contains trusted SSL CA certificate files.

- `MYSQL_OPT_SSL_CERT` (argument type: `char *`)

  The path name of the client public key certificate file.

- `MYSQL_OPT_SSL_CIPHER` (argument type: `char *`)

  The list of permissible ciphers for SSL encryption.

- `MYSQL_OPT_SSL_CRL` (argument type: `char *`)

The path name of the file containing certificate revocation lists.

- `MYSQL_OPT_SSL_CRLPATH` (argument type: `char *`)

  The path name of the directory that contains files containing certificate revocation lists.

- `MYSQL_OPT_SSL_FIPS_MODE` (argument type: `unsigned int *`)

  Controls whether to enable FIPS mode on the client side. The `MYSQL_OPT_SSL_FIPS_MODE` option differs from other `MYSQL_OPT_SSL_xxx` options in that it is not used to establish encrypted connections, but rather to affect which cryptographic operations to permit. See FIPS Support.

  Permitted option values are `SSL_FIPS_MODE_OFF`, `SSL_FIPS_MODE_ON`, and `SSL_FIPS_MODE_STRICT`.

  > **Note**
  >
  > If the OpenSSL FIPS Object Module is not available, the only permitted value for `MYSQL_OPT_SSL_FIPS_MODE` is `SSL_FIPS_MODE_OFF`. In this case, setting `MYSQL_OPT_SSL_FIPS_MODE` to `SSL_FIPS_MODE_ON` or `SSL_FIPS_MODE_STRICT` causes the client to produce a warning at startup and to operate in non-FIPS mode.

- `MYSQL_OPT_SSL_KEY` (argument type: `char *`)

  The path name of the client private key file.

- `MYSQL_OPT_SSL_MODE` (argument type: `unsigned int *`)

  The security state to use for the connection to the server: `SSL_MODE_DISABLED`, `SSL_MODE_PREFERRED`, `SSL_MODE_REQUIRED`, `SSL_MODE_VERIFY_CA`, `SSL_MODE_VERIFY_IDENTITY`. If this option is not specified, the default is `SSL_MODE_PREFERRED`. These modes are the permitted values of the `mysql_ssl_mode` enumeration defined in `mysql.h`. For more information about the security states, see the description of `--ssl-mode` in Command Options for Encrypted Connections.

- `MYSQL_OPT_TLS_CIPHERSUITES` (argument type: `char *`)

  Which ciphersuites the client permits for encrypted connections that use TLSv1.3. The value is a list of one or more colon-separated ciphersuite names. The ciphersuites that can be named for this option depend on the SSL library used to compile MySQL. For details, see Encrypted Connection TLS Protocols and Ciphers.

  This option was added in MySQL 8.0.16.

- `MYSQL_OPT_TLS_VERSION` (argument type: `char *`)

  Which protocols the client permits for encrypted connections. The value is a list of one or more comma-separated protocol versions. The protocols that can be named for this option depend on the SSL library used to compile MySQL. For details, see Encrypted Connection TLS Protocols and Ciphers.

- `MYSQL_OPT_USE_RESULT` (argument: not used)

  This option is unused.

- `MYSQL_OPT_WRITE_TIMEOUT` (argument type: `unsigned int *`)

  The timeout in seconds for each attempt to write to the server. There is a retry if necessary, so the total effective timeout value is two times the option value.

- `MYSQL_OPT_ZSTD_COMPRESSION_LEVEL` (argument type: `unsigned int *`)

  The compression level to use for connections to the server that use the `zstd` compression algorithm. The permitted levels are from 1 to 22, with larger values indicating increasing levels of compression. If this option is not specified, the default `zstd` compression level is 3. The compression level setting has no effect on connections that do not use `zstd` compression.

  For more information, see Connection Compression Control.

  This option was added in MySQL 8.0.18. For asynchronous operations, the option has no effect until MySQL 8.0.21.

- `MYSQL_PLUGIN_DIR` (argument type: `char *`)

  The directory in which to look for client plugins.

- `MYSQL_READ_DEFAULT_FILE` (argument type: `char *`)

  Read options from the named option file instead of from `my.cnf`.

- `MYSQL_READ_DEFAULT_GROUP` (argument type: `char *`)

  Read options from the named group from `my.cnf` or the file specified with `MYSQL_READ_DEFAULT_FILE`.

- `MYSQL_REPORT_DATA_TRUNCATION` (argument type: `bool *`)

  Enable or disable reporting of data truncation errors for prepared statements using the `error` member of `MYSQL_BIND` structures. (Default: enabled.)

- `MYSQL_SERVER_PUBLIC_KEY` (argument type: `char *`)

  The path name to a file in PEM format containing a client-side copy of the public key required by the server for RSA key pair-based password exchange. This option applies to clients that authenticate with the `sha256_password` or `caching_sha2_password` authentication plugin. This option is ignored for accounts that do not authenticate with one of those plugins. It is also ignored if RSA-based password exchange is not used, as is the case when the client connects to the server using a secure connection.

  If `MYSQL_SERVER_PUBLIC_KEY` is given and specifies a valid public key file, it takes precedence over `MYSQL_OPT_GET_SERVER_PUBLIC_KEY`.

  For information about the `sha256_password` and `caching_sha2_password` plugins, see SHA-256 Pluggable Authentication, and Caching SHA-2 Pluggable Authentication.

- `MYSQL_SET_CHARSET_DIR` (argument type: `char *`)

  The path name of the directory that contains character set definition files.

- `MYSQL_SET_CHARSET_NAME` (argument type: `char *`)

  The name of the character set to use as the default character set. The argument can be `MYSQL_AUTODETECT_CHARSET_NAME` to cause the character set to be autodetected based on the operating system setting (see Connection Character Sets and Collations).

- `MYSQL_SHARED_MEMORY_BASE_NAME` (argument type: `char *`)

  The name of the shared-memory object for communication to the server on Windows, if the server supports shared-memory connections. Specify the same value as used for the `shared_memory_base_name` system variable. of the `mysqld` server you want to connect to.

The `client` group is always read if you use `MYSQL_READ_DEFAULT_FILE` or `MYSQL_READ_DEFAULT_GROUP`.

The specified group in the option file may contain the following options.

Option	Description
`character-sets-dir=dir_name`	The directory where character sets are installed.
`compress`	Use the compressed client/server protocol.
`connect-timeout=seconds`	The connect timeout in seconds. On Linux this timeout is also used for waiting for the first answer from the server.
`database=db_name`	Connect to this database if no database was specified in the connect command.
`debug`	Debug options.
`default-character-set=charset_name`	The default character set to use.
`disable-local-infile`	Disable use of `LOAD DATA LOCAL`.
`enable-cleartext-plugin`	Enable the `mysql_clear_password` cleartext authentication plugin.
`host=host_name`	Default host name.
`init-command=stmt`	Statement to execute when connecting to MySQL server. Automatically re-executed if reconnection occurs.
`interactive-timeout=seconds`	Same as specifying `CLIENT_INTERACTIVE` to `mysql_real_connect()`. See Section 7.55, “mysql_real_connect()”.
`local-infile[={0|1}]`	If no argument or nonzero argument, enable use of `LOAD DATA LOCAL`; otherwise disable.
`max_allowed_packet=bytes`	Maximum size of packet that client can read from server.
`multi-queries`, `multi-results`	Enable multiple result sets from multiple-statement executions or stored procedures.
`multi-statements`	Enable the client to send multiple statements in a single string (separated by `;` characters).
`password=password`	Default password.
`pipe`	Use named pipes to connect to a MySQL server on Windows.
`port=port_num`	Default port number.
`protocol={TCP|SOCKET|PIPE|MEMORY}`	The protocol to use when connecting to the server.
`return-found-rows`	Tell `mysql_info()` to return found rows instead of updated rows when using `UPDATE`.
`shared-memory-base-name=name`	Shared-memory name to use to connect to server.

Option	Description
socket={*file_name*|*pipe_name*}	Default socket file.
ssl-ca=*file_name*	Certificate Authority file.
ssl-capath=*dir_name*	Certificate Authority directory.
ssl-cert=*file_name*	Certificate file.
ssl-cipher=*cipher_list*	Permissible SSL ciphers.
ssl-key=*file_name*	Key file.
timeout=*seconds*	Like connect-timeout.
user	Default user.

timeout has been replaced by connect-timeout, but timeout is still supported for backward compatibility.

For more information about option files used by MySQL programs, see Using Option Files.

Return Values

Zero for success. Nonzero if you specify an unknown option.

Example

The following mysql_options() calls request the use of compression in the client/server protocol, cause options to be read from the [odbc] group in option files, and disable transaction autocommit mode:

```
MYSQL mysql;

mysql_init(&mysql);
mysql_options(&mysql,MYSQL_OPT_COMPRESS,0);
mysql_options(&mysql,MYSQL_READ_DEFAULT_GROUP,"odbc");
mysql_options(&mysql,MYSQL_INIT_COMMAND,"SET autocommit=0");
if (!mysql_real_connect(&mysql,"host","user","passwd","database",0,NULL,0))
{
    fprintf(stderr, "Failed to connect to database: Error: %s\n",
          mysql_error(&mysql));
}
```

7.52 mysql_options4()

```
int
mysql_options4(MYSQL *mysql,
               enum mysql_option option,
               const void *arg1,
               const void *arg2)
```

Description

mysql_options4() is similar to mysql_options() but has an extra fourth argument so that two values can be passed for the option specified in the second argument.

The following list describes the permitted options, their effect, and how arg1 and arg2 are used.

- MYSQL_OPT_CONNECT_ATTR_ADD (argument types: char *, char *)

  This option adds an attribute key-value pair to the current set of connection attributes to pass to the server at connect time. Both arguments are pointers to null-terminated strings. The first and second

strings indicate the key and value, respectively. If the key is empty or already exists in the current set of connection attributes, an error occurs. Comparison of the key name with existing keys is case-sensitive.

Key names that begin with an underscore (_) are reserved for internal use and should not be created by application programs. This convention permits new attributes to be introduced by MySQL without colliding with application attributes.

`mysql_options4()` imposes a limit of 64KB on the aggregate size of connection attribute data it will accept. For calls that cause this limit to be exceeded, a `CR_INVALID_PARAMETER_NO` error occurs. Attribute size-limit checks also occur on the server side. For details, see Performance Schema Connection Attribute Tables, which also describes how the Performance Schema exposes connection attributes through the `session_connect_attrs` and `session_account_connect_attrs` tables.

See also the descriptions for the `MYSQL_OPT_CONNECT_ATTR_RESET` and `MYSQL_OPT_CONNECT_ATTR_DELETE` options in the description of the `mysql_options()` function.

Return Values

Zero for success. Nonzero if you specify an unknown option.

Errors

- `CR_DUPLICATE_CONNECTION_ATTR`

  A duplicate attribute name was specified.

- `CR_INVALID_PARAMETER_NO`

  A key name was empty or the amount of key-value connection attribute data exceeds 64KB limit.

- `CR_OUT_OF_MEMORY`

  Out of memory.

Example

This example demonstrates the calls that specify connection attributes:

```
MYSQL mysql;

mysql_init(&mysql);
mysql_options(&mysql,MYSQL_OPT_CONNECT_ATTR_RESET, 0);
mysql_options4(&mysql,MYSQL_OPT_CONNECT_ATTR_ADD, "key1", "value1");
mysql_options4(&mysql,MYSQL_OPT_CONNECT_ATTR_ADD, "key2", "value2");
mysql_options4(&mysql,MYSQL_OPT_CONNECT_ATTR_ADD, "key3", "value3");
mysql_options(&mysql,MYSQL_OPT_CONNECT_ATTR_DELETE, "key1");
if (!mysql_real_connect(&mysql,"host","user","passwd","database",0,NULL,0))
{
    fprintf(stderr, "Failed to connect to database: Error: %s\n",
        mysql_error(&mysql));
}
```

7.53 mysql_ping()

```
int
mysql_ping(MYSQL *mysql)
```

Description

Checks whether the connection to the server is working. If the connection has gone down and auto-reconnect is enabled an attempt to reconnect is made. If the connection is down and auto-reconnect is disabled, `mysql_ping()` returns an error.

Auto-reconnect is disabled by default. To enable it, call `mysql_options()` with the `MYSQL_OPT_RECONNECT` option. For details, see Section 7.51, "mysql_options()".

`mysql_ping()` can be used by clients that remain idle for a long while, to check whether the server has closed the connection and reconnect if necessary.

If `mysql_ping()`) does cause a reconnect, there is no explicit indication of it. To determine whether a reconnect occurs, call `mysql_thread_id()` to get the original connection identifier before calling `mysql_ping()`, then call `mysql_thread_id()` again to see whether the identifier has changed.

If reconnect occurs, some characteristics of the connection will have been reset. For details about these characteristics, see Chapter 28, *C API Automatic Reconnection Control*.

Return Values

Zero if the connection to the server is active. Nonzero if an error occurred. A nonzero return does not indicate whether the MySQL server itself is down; the connection might be broken for other reasons such as network problems.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

7.54 mysql_query()

```
int
mysql_query(MYSQL *mysql,
            const char *stmt_str)
```

Description

Executes the SQL statement pointed to by the null-terminated string `stmt_str`. Normally, the string must consist of a single SQL statement without a terminating semicolon (`;`) or `\g`. If multiple-statement execution has been enabled, the string can contain several statements separated by semicolons. See Chapter 23, *C API Multiple Statement Execution Support*.

`mysql_query()` cannot be used for statements that contain binary data; you must use `mysql_real_query()` instead. (Binary data may contain the `\0` character, which `mysql_query()` interprets as the end of the statement string.)

To determine whether a statement returns a result set, call `mysql_field_count()`. See Section 7.23, "mysql_field_count()".

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- CR_COMMANDS_OUT_OF_SYNC

  Commands were executed in an improper order.

- CR_SERVER_GONE_ERROR

  The MySQL server has gone away.

- CR_SERVER_LOST

  The connection to the server was lost during the query.

- CR_UNKNOWN_ERROR

  An unknown error occurred.

7.55 mysql_real_connect()

```
MYSQL *
mysql_real_connect(MYSQL *mysql,
                   const char *host,
                   const char *user,
                   const char *passwd,
                   const char *db,
                   unsigned int port,
                   const char *unix_socket,
                   unsigned long client_flag)
```

Description

> **Note**
>
> mysql_real_connect() is a synchronous function. Its asynchronous counterpart is mysql_real_connect_nonblocking(), for use by applications that require asynchronous communication with the server. See Chapter 12, *C API Asynchronous Interface*.
>
> To connect using a DNS SRV record, use mysql_real_connect_dns_srv(). See Section 7.56, “mysql_real_connect_dns_srv()”.

mysql_real_connect() attempts to establish a connection to a MySQL server running on host. Client programs must successfully connect to a server before executing any other API functions that require a valid MYSQL connection handler structure.

Specify the arguments as follows:

- For the first argument, specify the address of an existing MYSQL structure. Before calling mysql_real_connect(), call mysql_init() to initialize the MYSQL structure. You can change a lot of connect options with the mysql_options() call. See Section 7.51, “mysql_options()”.

- The value of host may be either a host name or an IP address. The client attempts to connect as follows:

- If `host` is `NULL` or the string `"localhost"`, a connection to the local host is assumed:
  - On Windows, the client connects using a shared-memory connection, if the server has shared-memory connections enabled.
  - On Unix, the client connects using a Unix socket file. The `unix_socket` argument or the `MYSQL_UNIX_PORT` environment variable may be used to specify the socket name.
- On Windows, if `host` is `"."`, or TCP/IP is not enabled and no `unix_socket` is specified or the host is empty, the client connects using a named pipe, if the server has named-pipe connections enabled. If named-pipe connections are not enabled, an error occurs.
- Otherwise, TCP/IP is used.

You can also influence the type of connection to use with the `MYSQL_OPT_PROTOCOL` or `MYSQL_OPT_NAMED_PIPE` options to `mysql_options()`. The type of connection must be supported by the server.

- The `user` argument contains the user's MySQL login ID. If `user` is `NULL` or the empty string `""`, the current user is assumed. Under Unix, this is the current login name. Under Windows ODBC, the current user name must be specified explicitly. See the Connector/ODBC section of Connectors and APIs.
- The `passwd` argument contains the password for `user`. If `passwd` is `NULL`, only entries in the `user` table for the user that have a blank (empty) password field are checked for a match. This enables the database administrator to set up the MySQL privilege system in such a way that users get different privileges depending on whether they have specified a password.

  > **Note**
  >
  > Do not attempt to encrypt the password before calling `mysql_real_connect()`; password encryption is handled automatically by the client API.

- The `user` and `passwd` arguments use whatever character set has been configured for the `MYSQL` object. By default, this is `utf8mb4`, but can be changed by calling `mysql_options(mysql, MYSQL_SET_CHARSET_NAME, "charset_name")` prior to connecting.
- `db` is the database name. If `db` is not `NULL`, the connection sets the default database to this value.
- If `port` is not 0, the value is used as the port number for the TCP/IP connection. Note that the `host` argument determines the type of the connection.
- If `unix_socket` is not `NULL`, the string specifies the socket or named pipe to use. Note that the `host` argument determines the type of the connection.
- The value of `client_flag` is usually 0, but can be set to a combination of the following flags to enable certain features:
  - `CAN_HANDLE_EXPIRED_PASSWORDS`: The client can handle expired passwords. For more information, see Server Handling of Expired Passwords.
  - `CLIENT_COMPRESS`: Use compression in the client/server protocol.
  - `CLIENT_FOUND_ROWS`: Return the number of found (matched) rows, not the number of changed rows.
  - `CLIENT_IGNORE_SIGPIPE`: Prevents the client library from installing a `SIGPIPE` signal handler. This can be used to avoid conflicts with a handler that the application has already installed.

- CLIENT_IGNORE_SPACE: Permit spaces after function names. Makes all functions names reserved words.
- CLIENT_INTERACTIVE: Permit interactive_timeout seconds of inactivity (rather than wait_timeout seconds) before closing the connection. The client's session wait_timeout variable is set to the value of the session interactive_timeout variable.
- CLIENT_LOCAL_FILES: Enable LOAD DATA LOCAL handling.
- CLIENT_MULTI_RESULTS: Tell the server that the client can handle multiple result sets from multiple-statement executions or stored procedures. This flag is automatically enabled if CLIENT_MULTI_STATEMENTS is enabled. See the note following this table for more information about this flag.
- CLIENT_MULTI_STATEMENTS: Tell the server that the client may send multiple statements in a single string (separated by ; characters). If this flag is not set, multiple-statement execution is disabled. See the note following this table for more information about this flag.
- CLIENT_NO_SCHEMA Do not permit *db_name.tbl_name.col_name* syntax. This is for ODBC. It causes the parser to generate an error if you use that syntax, which is useful for trapping bugs in some ODBC programs.
- CLIENT_ODBC: Unused.
- CLIENT_OPTIONAL_RESULTSET_METADATA: This flag makes result set metadata optional. Suppression of metadata transfer can improve performance, particularly for sessions that execute many queries that return few rows each. For details about managing result set metadata transfer, see Chapter 27, *C API Optional Result Set Metadata*.
- CLIENT_SSL: Use SSL (encrypted protocol). Do not set this option within an application program; it is set internally in the client library. Instead, use mysql_options() or mysql_ssl_set() before calling mysql_real_connect().
- CLIENT_REMEMBER_OPTIONS Remember options specified by calls to mysql_options(). Without this option, if mysql_real_connect() fails, you must repeat the mysql_options() calls before trying to connect again. With this option, the mysql_options() calls need not be repeated.

If your program uses CALL statements to execute stored procedures, the CLIENT_MULTI_RESULTS flag must be enabled. This is because each CALL returns a result to indicate the call status, in addition to any result sets that might be returned by statements executed within the procedure. Because CALL can return multiple results, process them using a loop that calls mysql_next_result() to determine whether there are more results.

CLIENT_MULTI_RESULTS can be enabled when you call mysql_real_connect(), either explicitly by passing the CLIENT_MULTI_RESULTS flag itself, or implicitly by passing CLIENT_MULTI_STATEMENTS (which also enables CLIENT_MULTI_RESULTS). CLIENT_MULTI_RESULTS is enabled by default.

If you enable CLIENT_MULTI_STATEMENTS or CLIENT_MULTI_RESULTS, process the result for every call to mysql_query() or mysql_real_query() by using a loop that calls mysql_next_result() to determine whether there are more results. For an example, see Chapter 23, *C API Multiple Statement Execution Support*.

For some arguments, it is possible to have the value taken from an option file rather than from an explicit value in the mysql_real_connect() call. To do this, call mysql_options() with the MYSQL_READ_DEFAULT_FILE or MYSQL_READ_DEFAULT_GROUP option before calling

mysql_real_connect(). Then, in the mysql_real_connect() call, specify the “no-value” value for each argument to be read from an option file:

- For host, specify a value of NULL or the empty string ("").
- For user, specify a value of NULL or the empty string.
- For passwd, specify a value of NULL. (For the password, a value of the empty string in the mysql_real_connect() call cannot be overridden in an option file, because the empty string indicates explicitly that the MySQL account must have an empty password.)
- For db, specify a value of NULL or the empty string.
- For port, specify a value of 0.
- For unix_socket, specify a value of NULL.

If no value is found in an option file for an argument, its default value is used as indicated in the descriptions given earlier in this section.

Return Values

A MYSQL* connection handler if the connection was successful, NULL if the connection was unsuccessful. For a successful connection, the return value is the same as the value of the first argument.

Errors

- CR_CONN_HOST_ERROR

  Failed to connect to the MySQL server.

- CR_CONNECTION_ERROR

  Failed to connect to the local MySQL server.

- CR_IPSOCK_ERROR

  Failed to create an IP socket.

- CR_OUT_OF_MEMORY

  Out of memory.

- CR_SOCKET_CREATE_ERROR

  Failed to create a Unix socket.

- CR_UNKNOWN_HOST

  Failed to find the IP address for the host name.

- CR_VERSION_ERROR

  A protocol mismatch resulted from attempting to connect to a server with a client library that uses a different protocol version.

- CR_NAMEDPIPEOPEN_ERROR

Failed to create a named pipe on Windows.

- CR_NAMEDPIPEWAIT_ERROR

  Failed to wait for a named pipe on Windows.

- CR_NAMEDPIPESETSTATE_ERROR

  Failed to get a pipe handler on Windows.

- CR_SERVER_LOST

  If connect_timeout > 0 and it took longer than connect_timeout seconds to connect to the server or if the server died while executing the init-command.

- CR_ALREADY_CONNECTED

  The MYSQL connection handler is already connected.

Example

```
MYSQL mysql;

mysql_init(&mysql);
mysql_options(&mysql,MYSQL_READ_DEFAULT_GROUP,"your_prog_name");
if (!mysql_real_connect(&mysql,"host","user","passwd","database",0,NULL,0))
{
    fprintf(stderr, "Failed to connect to database: Error: %s\n",
          mysql_error(&mysql));
}
```

By using mysql_options() the MySQL client library reads the [client] and [your_prog_name] sections in the my.cnf file. This enables you to add options to the [your_prog_name] section to ensure that your program works, even if someone has set up MySQL in some nonstandard way.

7.56 mysql_real_connect_dns_srv()

```
MYSQL *
mysql_real_connect_dns_srv(MYSQL *mysql,
                           const char *dns_srv_name,
                           const char *user,
                           const char *passwd,
                           const char *db,
                           unsigned long client_flag)
```

Description

> **Note**
>
> mysql_real_connect_dns_srv() is a synchronous function. Unlike mysql_real_connect(), it has no asynchronous counterpart.

mysql_real_connect_dns_srv() is similar to mysql_real_connect(), except that the argument list does not specify the particular host of the MySQL server to connect to. Instead, it names a DNS SRV record that specifies a group of servers. For information about DNS SRV support in MySQL, see Connecting to the Server Using DNS SRV Records.

The dns_srv_name argument for mysql_real_connect_dns_srv() takes the place of the host, port, and unix_socket arguments for mysql_real_connect(). The dns_srv_name argument

names a DNS SRV record that determines the candidate hosts to use for establishing a connection to a MySQL server.

The `mysql`, `user`, `passwd`, `db`, and `client_flag` arguments to `mysql_real_connect_dns_srv()` have the same meanings as for `mysql_real_connect()`. For descriptions of their meanings, see Section 7.55, “mysql_real_connect()”.

Suppose that DNS is configured with this SRV information for the `example.com` domain:

```
Name                     TTL   Class   Priority Weight Port Target
_mysql._tcp.example.com. 86400 IN SRV  0        5      3306 host1.example.com
_mysql._tcp.example.com. 86400 IN SRV  0        10     3306 host2.example.com
_mysql._tcp.example.com. 86400 IN SRV  10       5      3306 host3.example.com
_mysql._tcp.example.com. 86400 IN SRV  20       5      3306 host4.example.com
```

To use that DNS SRV record, pass `"_mysql._tcp.example.com"` as the `dns_srv_name` argument to `mysql_real_connect_dns_srv()`, which then attempts a connection to each server in the group until a successful connection is established. A failure to connect occurs only if a connection cannot be established to any of the servers. The priority and weight values in the DNS SRV record determine the order in which servers should be tried.

`mysql_real_connect_dns_srv()` attempts to establish TCP connections only.

The client library performs a DNS SRV lookup for each call to `mysql_real_connect_dns_srv()`. The client library does no caching of lookup results.

Return Values

A `MYSQL*` connection handler if the connection was successful, `NULL` if the connection was unsuccessful. For a successful connection, the return value is the same as the value of the first argument.

Errors

The same that you can get from `mysql_real_connect()`, plus:

- `CR_DNS_SRV_LOOKUP_FAILED`

  DNS SRV lookup failed.

Example

The following example uses the name of the DNS SRV record shown previously as the source of candidate servers for establishing a connection.

```
MYSQL mysql;
const char *dns_srv_name = "_mysql._tcp.example.com";

mysql_init(&mysql);
if (!mysql_real_connect_dns_srv(&mysql,dns_srv_name,"user","passwd","database",0))
{
    fprintf(stderr, "Failed to connect to database: Error: %s\n",
          mysql_error(&mysql));
}
```

7.57 mysql_real_escape_string()

```
unsigned long
mysql_real_escape_string(MYSQL *mysql,
```

```
                        char *to,
                        const char *from,
                        unsigned long length)
```

Description

This function creates a legal SQL string for use in an SQL statement. See String Literals.

> **Note**
>
> `mysql_real_escape_string()` fails and produces an `CR_INSECURE_API_ERR` error if the `NO_BACKSLASH_ESCAPES` SQL mode is enabled. In this case, the function cannot escape quote characters except by doubling them, and to do this properly, it must know more information about the quoting context than is available. Instead, use `mysql_real_escape_string_quote()`, which takes an extra argument for specifying the quoting context.

The `mysql` argument must be a valid, open connection because character escaping depends on the character set in use by the server.

The string in the `from` argument is encoded to produce an escaped SQL string, taking into account the current character set of the connection. The result is placed in the `to` argument, followed by a terminating null byte.

Characters encoded are `\`, `'`, `"`, `NUL` (ASCII 0), `\n`, `\r`, and Control+Z. Strictly speaking, MySQL requires only that backslash and the quote character used to quote the string in the query be escaped. `mysql_real_escape_string()` quotes the other characters to make them easier to read in log files. For comparison, see the quoting rules for literal strings and the `QUOTE()` SQL function in String Literals, and String Functions and Operators.

The string pointed to by `from` must be `length` bytes long. You must allocate the `to` buffer to be at least `length*2+1` bytes long. (In the worst case, each character may need to be encoded as using two bytes, and there must be room for the terminating null byte.) When `mysql_real_escape_string()` returns, the contents of `to` is a null-terminated string. The return value is the length of the encoded string, not including the terminating null byte.

If you must change the character set of the connection, use the `mysql_set_character_set()` function rather than executing a `SET NAMES` (or `SET CHARACTER SET`) statement. `mysql_set_character_set()` works like `SET NAMES` but also affects the character set used by `mysql_real_escape_string()`, which `SET NAMES` does not.

Example

The following example inserts two escaped strings into an `INSERT` statement, each within single quote characters:

```
char query[1000],*end;

end = my_stpcpy(query,"INSERT INTO test_table VALUES('");
end += mysql_real_escape_string(&mysql,end,"What is this",12);
end = my_stpcpy(end,"','");
end += mysql_real_escape_string(&mysql,end,"binary data: \0\r\n",16);
end = my_stpcpy(end,"')");

if (mysql_real_query(&mysql,query,(unsigned int) (end - query)))
{
   fprintf(stderr, "Failed to insert row, Error: %s\n",
           mysql_error(&mysql));
```

```
}
```

The `my_stpcpy()` function used in the example is included in the `libmysqlclient` library and works like `strcpy()` but returns a pointer to the terminating null of the first parameter.

Return Values

The length of the encoded string that is placed into the `to` argument, not including the terminating null byte, or -1 if an error occurs.

Because `mysql_real_escape_string()` returns an unsigned value, you can check for -1 by comparing the return value to `(unsigned long)-1` (or to `(unsigned long)~0`, which is equivalent).

Errors

- `CR_INSECURE_API_ERR`

  This error occurs if the `NO_BACKSLASH_ESCAPES` SQL mode is enabled because, in that case, `mysql_real_escape_string()` cannot be guaranteed to produce a properly encoded result. To avoid this error, use `mysql_real_escape_string_quote()` instead.

7.58 mysql_real_escape_string_quote()

```
unsigned long
mysql_real_escape_string_quote(MYSQL *mysql,
                               char *to,
                               const char *from,
                               unsigned long length,
                               char quote)
```

Description

This function creates a legal SQL string for use in an SQL statement. See String Literals.

The `mysql` argument must be a valid, open connection because character escaping depends on the character set in use by the server.

The string in the `from` argument is encoded to produce an escaped SQL string, taking into account the current character set of the connection. The result is placed in the `to` argument, followed by a terminating null byte.

Characters encoded are `\`, `'`, `"`, `NUL` (ASCII 0), `\n`, `\r`, Control+Z, and `` ` ``. Strictly speaking, MySQL requires only that backslash and the quote character used to quote the string in the query be escaped. `mysql_real_escape_string_quote()` quotes the other characters to make them easier to read in log files. For comparison, see the quoting rules for literal strings and the `QUOTE()` SQL function in String Literals, and String Functions and Operators.

> **Note**
>
> If the `ANSI_QUOTES` SQL mode is enabled, `mysql_real_escape_string_quote()` cannot be used to escape double quote characters for use within double-quoted identifiers. (The function cannot tell whether the mode is enabled to determine the proper escaping character.)

The string pointed to by `from` must be `length` bytes long. You must allocate the `to` buffer to be at least `length*2+1` bytes long. (In the worst case, each character may need to be encoded as using two bytes, and there must be room for the terminating null byte.) When `mysql_real_escape_string_quote()`

returns, the contents of `to` is a null-terminated string. The return value is the length of the encoded string, not including the terminating null byte.

The `quote` argument indicates the context in which the escaped string is to be placed. Suppose that you intend to escape the `from` argument and insert the escaped string (designated here by *`str`*) into one of the following statements:

```
1) SELECT * FROM table WHERE name = 'str'
2) SELECT * FROM table WHERE name = "str"
3) SELECT * FROM `str` WHERE id = 103
```

To perform escaping properly for each statement, call `mysql_real_escape_string_quote()` as follows, where the final argument indicates the quoting context:

```
1) len = mysql_real_escape_string_quote(&mysql,to,from,from_len,'\'');
2) len = mysql_real_escape_string_quote(&mysql,to,from,from_len,'"');
3) len = mysql_real_escape_string_quote(&mysql,to,from,from_len,'`');
```

If you must change the character set of the connection, use the `mysql_set_character_set()` function rather than executing a `SET NAMES` (or `SET CHARACTER SET`) statement. `mysql_set_character_set()` works like `SET NAMES` but also affects the character set used by `mysql_real_escape_string_quote()`, which `SET NAMES` does not.

Example

The following example inserts two escaped strings into an `INSERT` statement, each within single quote characters:

```
char query[1000],*end;

end = my_stpcpy(query,"INSERT INTO test_table VALUES('");
end += mysql_real_escape_string_quote(&mysql,end,"What is this",12,'\'');
end = my_stpcpy(end,"','");
end += mysql_real_escape_string_quote(&mysql,end,"binary data: \0\r\n",16,'\'');
end = my_stpcpy(end,"')");

if (mysql_real_query(&mysql,query,(unsigned int) (end - query)))
{
   fprintf(stderr, "Failed to insert row, Error: %s\n",
           mysql_error(&mysql));
}
```

The `my_stpcpy()` function used in the example is included in the `libmysqlclient` library and works like `strcpy()` but returns a pointer to the terminating null of the first parameter.

Return Values

The length of the encoded string that is placed into the `to` argument, not including the terminating null byte.

Errors

None.

7.59 mysql_real_query()

```
int
mysql_real_query(MYSQL *mysql,
                 const char *stmt_str,
                 unsigned long length)
```

Description

> **Note**
>
> `mysql_real_query()` is a synchronous function. Its asynchronous counterpart is `mysql_real_query_nonblocking()`, for use by applications that require asynchronous communication with the server. See Chapter 12, *C API Asynchronous Interface*.

`mysql_real_query()` executes the SQL statement pointed to by `stmt_str`, a string `length` bytes long. Normally, the string must consist of a single SQL statement without a terminating semicolon (`;`) or `\g`. If multiple-statement execution has been enabled, the string can contain several statements separated by semicolons. See Chapter 23, *C API Multiple Statement Execution Support*.

`mysql_query()` cannot be used for statements that contain binary data; you must use `mysql_real_query()` instead. (Binary data may contain the `\0` character, which `mysql_query()` interprets as the end of the statement string.) In addition, `mysql_real_query()` is faster than `mysql_query()` because it does not call `strlen()` on the statement string.

To determine whether a statement returns a result set, call `mysql_field_count()`. See Section 7.23, “mysql_field_count()”.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

7.60 mysql_refresh()

```
int
mysql_refresh(MYSQL *mysql,
              unsigned int options)
```

Description

> **Note**
>
> `mysql_refresh()` is deprecated and will be removed in a future version of MySQL. Instead, use `mysql_query()` to execute a `FLUSH` statement.

This function flushes tables or caches, or resets replication server information. The connected user must have the `RELOAD` privilege.

The `options` argument is a bitmask composed from any combination of the following values. Multiple values can be OR'ed together to perform multiple operations with a single call.

- `REFRESH_GRANT`

  Refresh the grant tables, like `FLUSH PRIVILEGES`.

- `REFRESH_LOG`

  Flush the logs, like `FLUSH LOGS`.

- `REFRESH_TABLES`

  Flush the table cache, like `FLUSH TABLES`.

- `REFRESH_HOSTS`

  Flush the host cache, like `FLUSH HOSTS`.

- `REFRESH_STATUS`

  Reset status variables, like `FLUSH STATUS`.

- `REFRESH_THREADS`

  Flush the thread cache.

- `REFRESH_SLAVE`

  On a replica server, reset the source server information and restart the replica, like `RESET SLAVE`.

- `REFRESH_MASTER`

  On a source server, remove the binary log files listed in the binary log index and truncate the index file, like `RESET MASTER`.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

An unknown error occurred.

7.61 mysql_reload()

```
int
mysql_reload(MYSQL *mysql)
```

Description

Asks the MySQL server to reload the grant tables. The connected user must have the `RELOAD` privilege.

This function is deprecated. Use `mysql_query()` to issue an SQL `FLUSH PRIVILEGES` statement instead.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

7.62 mysql_reset_connection()

```
int
mysql_reset_connection(MYSQL *mysql)
```

Description

Resets the connection to clear the session state.

`mysql_reset_connection()` has effects similar to `mysql_change_user()` or an auto-reconnect except that the connection is not closed and reopened, and reauthentication is not done. The write set session history is reset. See Section 7.4, “mysql_change_user()”, and Chapter 28, *C API Automatic Reconnection Control*.

`mysql_reset_connection()` affects the connection-related state as follows:

- Rolls back any active transactions and resets autocommit mode.
- Releases all table locks.

- Closes (and drops) all TEMPORARY tables.
- Reinitializes session system variables to the values of the corresponding global system variables, including system variables that are set implicitly by statements such as SET NAMES.
- Loses user-defined variable settings.
- Releases prepared statements.
- Closes HANDLER variables.
- Resets the value of LAST_INSERT_ID() to 0.
- Releases locks acquired with GET_LOCK().
- Clears any current query attributes defined as a result of calling mysql_bind_param().

Return Values

Zero for success. Nonzero if an error occurred.

7.63 mysql_reset_server_public_key()

```
void
mysql_reset_server_public_key(void)
```

Description

Clears from the client library any cached copy of the public key required by the server for RSA key pair-based password exchange. This might be necessary when the server has been restarted with a different RSA key pair after the client program had called mysql_options() with the MYSQL_SERVER_PUBLIC_KEY option to specify the RSA public key. In such cases, connection failure can occur due to key mismatch. To fix this problem, the client can use either of the following approaches:

- The client can call mysql_reset_server_public_key() to clear the cached key and try again, after the public key file on the client side has been replaced with a file containing the new public key.
- The client can call mysql_reset_server_public_key() to clear the cached key, then call mysql_options() with the MYSQL_OPT_GET_SERVER_PUBLIC_KEY option (instead of MYSQL_SERVER_PUBLIC_KEY) to request the required public key from the server Do not use both MYSQL_OPT_GET_SERVER_PUBLIC_KEY and MYSQL_SERVER_PUBLIC_KEY because in that case, MYSQL_SERVER_PUBLIC_KEY takes precedence.

Return Values

None.

Errors

None.

7.64 mysql_result_metadata()

```
enum enum_resultset_metadata
mysql_result_metadata(MYSQL_RES *result)
```

Description

mysql_result_metadata() returns a value that indicates whether a result set has metadata. It can be useful for metadata-optional connections when the client does not know in advance whether particular result sets have metadata. For example, if a client executes a stored procedure that returns multiple result sets and might change the resultset_metadata system variable, the client can invoke mysql_result_metadata() for each result set to determine whether it has metadata.

For details about managing result set metadata transfer, see Chapter 27, *C API Optional Result Set Metadata*.

Return Values

mysql_result_metadata() returns one of these values:

```
enum enum_resultset_metadata {
 RESULTSET_METADATA_NONE= 0,
 RESULTSET_METADATA_FULL= 1
};
```

7.65 mysql_rollback()

```
bool
mysql_rollback(MYSQL *mysql)
```

Description

Rolls back the current transaction.

The action of this function is subject to the value of the completion_type system variable. In particular, if the value of completion_type is RELEASE (or 2), the server performs a release after terminating a transaction and closes the client connection. Call mysql_close() from the client program to close the connection from the client side.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

None.

7.66 mysql_row_seek()

```
MYSQL_ROW_OFFSET
mysql_row_seek(MYSQL_RES *result,
               MYSQL_ROW_OFFSET offset)
```

Description

Sets the row cursor to an arbitrary row in a query result set. The offset value is a row offset, typically a value returned from mysql_row_tell() or from mysql_row_seek(). This value is not a row number; to seek to a row within a result set by number, use mysql_data_seek() instead.

This function requires that the result set structure contains the entire result of the query, so mysql_row_seek() may be used only in conjunction with mysql_store_result(), not with mysql_use_result().

Return Values

The previous value of the row cursor. This value may be passed to a subsequent call to `mysql_row_seek()`.

Errors

None.

7.67 mysql_row_tell()

```
MYSQL_ROW_OFFSET
mysql_row_tell(MYSQL_RES *result)
```

Description

Returns the current position of the row cursor for the last `mysql_fetch_row()`. This value can be used as an argument to `mysql_row_seek()`.

Use `mysql_row_tell()` only after `mysql_store_result()`, not after `mysql_use_result()`.

Return Values

The current offset of the row cursor.

Errors

None.

7.68 mysql_select_db()

```
int
mysql_select_db(MYSQL *mysql,
                const char *db)
```

Description

Causes the database specified by `db` to become the default (current) database on the connection specified by `mysql`. In subsequent queries, this database is the default for table references that include no explicit database specifier.

`mysql_select_db()` fails unless the connected user can be authenticated as having permission to use the database or some object within it.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- CR_SERVER_LOST

  The connection to the server was lost during the query.

- CR_UNKNOWN_ERROR

  An unknown error occurred.

7.69 mysql_server_end()

```
void
mysql_server_end(void)
```

Description

This function finalizes the MySQL client library, which should be done when you are done using the library. However, mysql_server_end() is deprecated and mysql_library_end() should be used instead. See Section 7.41, “mysql_library_end()”.

> **Note**
>
> To avoid memory leaks after the application is done using the library (for example, after closing the connection to the server), be sure to call mysql_server_end() (or mysql_library_end()) explicitly. This enables memory managment to be performed to clean up and free resources used by the library.

Return Values

None.

7.70 mysql_server_init()

```
int
mysql_server_init(int argc,
                  char **argv,
                  char **groups)
```

Description

This function initializes the MySQL client library, which must be done before you call any other MySQL function. However, mysql_server_init() is deprecated and you should call mysql_library_init() instead. See Section 7.42, “mysql_library_init()”.

> **Note**
>
> To avoid memory leaks after the application is done using the library (for example, after closing the connection to the server), be sure to call mysql_server_end() (or mysql_library_end()) explicitly. This enables memory managment to be performed to clean up and free resources used by the library. See Section 7.41, “mysql_library_end()”.

Return Values

Zero for success. Nonzero if an error occurred.

7.71 mysql_session_track_get_first()

```
int
mysql_session_track_get_first(MYSQL *mysql,
                              enum enum_session_state_type type,
                              const char **data,
                              size_t *length)
```

Description

MySQL implements a session tracker mechanism whereby the server returns information about session state changes to clients. To control which notifications the server provides about state changes, client applications set system variables having names of the form `session_track_xxx`, such as `session_track_state_change`, `session_track_schema`, and `session_track_system_variables`. See Server Tracking of Client Session State Changes.

Change notification occurs in the MySQL client/server protocol, which includes tracker information in OK packets so that session state changes can be detected. To enable client applications to extract state-change information from OK packets, the MySQL C API provides a pair of functions:

- `mysql_session_track_get_first()` fetches the first part of the state-change information received from the server.
- `mysql_session_track_get_next()` fetches any remaining state-change information received from the server. Following a successful call to `mysql_session_track_get_first()`, call this function repeatedly as long as it returns success.

The `mysql_session_track_get_first()` parameters are used as follows. These descriptions also apply to `mysql_session_track_get_next()`, which takes the same parameters.

- `mysql`: The connection handler.
- `type`: The tracker type indicating what kind of information to retrieve. Permitted tracker values are the members of the `enum_session_state_type` enumeration defined in `mysql_com.h`:

  ```
  enum enum_session_state_type
  {
    SESSION_TRACK_SYSTEM_VARIABLES,            /* Session system variables */
    SESSION_TRACK_SCHEMA,                      /* Current schema */
    SESSION_TRACK_STATE_CHANGE,                /* Session state changes */
    SESSION_TRACK_GTIDS,                       /* GTIDs */
    SESSION_TRACK_TRANSACTION_CHARACTERISTICS, /* Transaction characteristics */
    SESSION_TRACK_TRANSACTION_STATE            /* Transaction state */
  };
  ```

  The members of that enumeration may change over time as MySQL implements additional session-information trackers. To make it easy for applications to loop over all possible tracker types regardless of the number of members, the `SESSION_TRACK_BEGIN` and `SESSION_TRACK_END` symbols are defined to be equal to the first and last members of the `enum_session_state_type` enumeration. The example code shown later in this section demonstrates this technique. (Of course, if the enumeration members change, you must recompile your application to enable it to take account of new trackers.)

- `data`: The address of a `const char *` variable. Following a successful call, this variable points to the returned data, which should be considered read only.
- `length`: The address of a `size_t` variable. Following a successful call, this variable contains the length of the data pointed to by the `data` parameter.

The following discussion describes how to interpret the `data` and `length` values according to the `type` value. It also indicates which system variable enables notifications for each tracker type.

- `SESSION_TRACK_SCHEMA`: This tracker type indicates that the default schema has been set. `data` is a string containing the new default schema name. `length` is the string length.

  To enable notifications for this tracker type, enable the `session_track_schema` system variable.

- `SESSION_TRACK_SYSTEM_VARIABLES`: This tracker type indicates that one or more tracked session system variables have been assigned a value. When a session system variable is assigned, two values per variable are returned (in separate calls). For the first call, `data` is a string containing the variable name and `length` is the string length. For the second call, `data` is a string containing the variable value and `length` is the string length.

  By default, notification is enabled for these session system variables:

  - `autocommit`
  - `character_set_client`
  - `character_set_connection`
  - `character_set_results`
  - `time_zone`

  To change the default notification for this tracker type, set the `session_track_schema` system variable to a list of comma-separated variables for which to track changes, or `*` to track changes for all variables. To disable notification of session variable assignments, set `session_track_system_variables` to the empty string.

- `SESSION_TRACK_STATE_CHANGE`: This tracker type indicates a change to some tracked attribute of session state. `data` is a byte containing a boolean flag that indicates whether session state changes occurred. `length` should be 1. The flag is represented as an ASCII value, not a binary (for example, `'1'`, not `0x01`).

  To enable notifications for this tracker type, enable the `session_track_state_change` system variable.

  This tracker reports changes for these attributes of session state:

  - The default schema (database).
  - Session-specific values for system variables.
  - User-defined variables.
  - Temporary tables.
  - Prepared statements.

- `SESSION_TRACK_GTIDS`: This tracker type indicates that GTIDs are available. `data` contains the GTID string. `length` is the string length. The GTID string is in the standard format for specifying a set of GTID values; see GTID Sets.

  To enable notifications for this tracker type, set the `session_track_gtids` system variable.

- `SESSION_TRACK_TRANSACTION_CHARACTERISTICS`: This tracker type indicates that transaction characteristics are available. `data` is a string containing the characteristics data. `length` is the string length. The characteristics tracker data string may be empty, or it may contain one or more SQL statements, each terminated by a semicolon:

- If no characteristics apply, the string is empty. The session defaults apply. (For isolation level and access mode, these defaults are given by the session values of the `transaction_isolation` and `transaction_read_only` system variables.)
- If a transaction was explicitly started, the string contains the statement or statements required to restart the transaction with the same characteristics. As a general rule, this is a `START TRANSACTION` statement (possibly with one or more of `READ ONLY`, `READ WRITE`, and `WITH CONSISTENT SNAPSHOT`). If any characteristics apply that cannot be passed to `START TRANSACTION`, such as `ISOLATION LEVEL`, a suitable `SET TRANSACTION` statement is prepended (for example, `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; START TRANSACTION READ WRITE;`).
- If a transaction was not explicitly started, but one-shot characteristics that apply only to the next transaction were set up, a `SET TRANSACTION` statement suitable for replicating that setup is generated (for example, `SET TRANSACTION READ ONLY;`).

  Next-transaction characteristics can be set using `SET TRANSACTION` without any `GLOBAL` or `SESSION` keyword, or by setting the `transaction_isolation` and `transaction_read_only` system variables using the syntax that applies only to the next transaction:

  ```
  SET @@transaction_isolation = value;
  SET @@transaction_read_only = value;
  ```

  For more information about transaction characteristic scope levels and how they are set, see Transaction Characteristic Scope.

To enable notifications for this tracker type, set the `session_track_transaction_info` system variable to `CHARACTERISTICS` (which also enables the `SESSION_TRACK_TRANSACTION_STATE` tracker type).

Transaction characteristics tracking enables the client to determine how to restart a transaction in another session so it has the same characteristics as in the original session.

Because characteristics may be set using `SET TRANSACTION` before a transaction is started, it is not safe for the client to assume that there are no transaction characteristics if no transaction is active. It is therefore unsafe not to track transaction characteristics and just switch the connection when no transaction is active (whether this is detected by the transaction state tracker or the traditional `SERVER_STATUS_IN_TRANS` flag). A client *must* subscribe to the transaction characteristics tracker if it may wish to switch its session to another connection at some point and transactions may be used.

The characteristics tracker tracks changes to the one-shot characteristics that apply only to the next transaction. It does not track changes to the session variables. Therefore, the client additionally must track the `transaction_isolation` and `transaction_read_only` system variables to correctly determine the session defaults that apply when next-transaction characteristic values are empty. (To track these variables, list them in the value of the `session_track_system_variables` system variable.)

- `SESSION_TRACK_TRANSACTION_STATE`: This tracker type indicates that transaction state information is available. `data` is a string containing ASCII characters, each of which indicates some aspect of the transaction state. `length` is the string length (always 8).

  To enable notifications for this tracker type, set the `session_track_transaction_info` system variable to `STATE`.

  Transaction state tracking enables the client to determine whether a transaction is in progress and whether it could be moved to a different session without being rolled back.

  The scope of the tracker item is the transaction. All state-indicating flags persist until the transaction is committed or rolled back. As statements are added to the transaction, additional flags may be set in successive tracker data values. However, no flags are cleared until the transaction ends.

  Transaction state is reported as a string containing a sequence of ASCII characters. Each active state has a unique character assigned to it as well as a fixed position in the sequence. The following list describes the permitted values for positions 1 through 8 of the sequence:

  - Position 1: Whether an active transaction is ongoing.
    - `T`: An explicitly started transaction is ongoing.
    - `I`: An implicitly started transaction (`autocommit=0`) is ongoing.
    - `_`: There is no active transaction.
  - Position 2: Whether nontransactional tables were read in the context of the current transaction.
    - `r`: One or more nontransactional tables were read.
    - `_`: No nontransactional tables were read so far.
  - Position 3: Whether transactional tables were read in the context of the current transaction.
    - `R`: One or more transactional tables were read.
    - `_`: No transactional tables were read so far.
  - Position 4: Whether unsafe writes (writes to nontransactional tables) were performed in the context of the current transaction.
    - `w`: One or more nontransactional tables were written.
    - `_`: No nontransactional tables were written so far.
  - Position 5: Whether any transactional tables were written in the context of the current transaction.
    - `W`: One or more transactional tables were written.
    - `_`: No transactional tables were written so far.
  - Position 6: Whether any unsafe statements were executed in the context of the current transaction. Statements containing nondeterministic constructs such as `RAND()` or `UUID()` are unsafe for statement-based replication.
    - `s`: One or more unsafe statements were executed.

  - _: No unsafe statements were executed so far.
- Position 7: Whether a result set was sent to the client during the current transaction.
  - S: A result set was sent.
  - _: No result sets were sent so far.
- Position 8: Whether a `LOCK TABLES` statement is in effect.
  - L: Tables are explicitly locked with `LOCK TABLES`.
  - _: `LOCK TABLES` is not active in the session.

Consider a session consisting of the following statements, including one to enable the transaction state tracker:

```
1. SET @@SESSION.session_track_transaction_info='STATE';
2. START TRANSACTION;
3. SELECT 1;
4. INSERT INTO t1 () VALUES();
5. INSERT INTO t1 () VALUES(1, RAND());
6. COMMIT;
```

With transaction state tracking enabled, the following `data` values result from those statements:

```
1. ________
2. T_______
3. T_____S_
4. T___W_S_
5. T___WsS_
6. ________
```

Return Values

Zero for success. Nonzero if an error occurred.

Errors

None.

Example

The following example shows how to call `mysql_session_track_get_first()` and `mysql_session_track_get_next()` to retrieve and display all available session state-change information following successful execution of an SQL statement string (represented by `stmt_str`). It is assumed that the application has set the `session_track_xxx` system variables that enable the notifications it wishes to receive.

```
printf("Execute: %s\n", stmt_str);

if (mysql_query(mysql, stmt_str) != 0)
{
  fprintf(stderr, "Error %u: %s\n",
          mysql_errno(mysql), mysql_error(mysql));
  return;
}

MYSQL_RES *result = mysql_store_result(mysql);
```

```
if (result) /* there is a result set to fetch */
{
  /* ... process rows here ... */
  printf("Number of rows returned: %lu\n",
          (unsigned long) mysql_num_rows(result));
  mysql_free_result(result);
}
else        /* there is no result set */
{
  if (mysql_field_count(mysql) == 0)
  {
    printf("Number of rows affected: %lu\n",
            (unsigned long) mysql_affected_rows(mysql));
  }
  else      /* an error occurred */
  {
    fprintf(stderr, "Error %u: %s\n",
             mysql_errno(mysql), mysql_error(mysql));
  }
}

/* extract any available session state-change information */
enum enum_session_state_type type;
for (type = SESSION_TRACK_BEGIN; type <= SESSION_TRACK_END; type++)
{
  const char *data;
  size_t length;

  if (mysql_session_track_get_first(mysql, type, &data, &length) == 0)
  {
    /* print info type and initial data */
    printf("Type=%d:\n", type);
    printf("mysql_session_track_get_first(): length=%d; data=%*.*s\n",
           (int) length, (int) length, (int) length, data);

    /* check for more data */
    while (mysql_session_track_get_next(mysql, type, &data, &length) == 0)
    {
      printf("mysql_session_track_get_next(): length=%d; data=%*.*s\n",
             (int) length, (int) length, (int) length, data);
    }
  }
}
```

7.72 mysql_session_track_get_next()

```
int
mysql_session_track_get_next(MYSQL *mysql,
                             enum enum_session_state_type type,
                             const char **data,
                             size_t *length)
```

Description

This function fetches additional session state-change information received from the server, following that retrieved by `mysql_session_track_get_first()`. The parameters for `mysql_session_track_get_next()` are the same as for `mysql_session_track_get_first()`.

Following a successful call to `mysql_session_track_get_first()`, call `mysql_session_track_get_next()` repeatedly until it returns nonzero to indicate no more information is available. The calling sequence for `mysql_session_track_get_next()` is similar to that for `mysql_session_track_get_first()`. For more information and an example that demonstrates both functions, see Section 7.71, “mysql_session_track_get_first()”.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

None.

7.73 mysql_set_character_set()

```
int
mysql_set_character_set(MYSQL *mysql,
                        const char *csname)
```

Description

This function is used to set the default character set for the current connection. The string `csname` specifies a valid character set name. The connection collation becomes the default collation of the character set. This function works like the `SET NAMES` statement, but also sets the value of `mysql->charset`, and thus affects the character set used by `mysql_real_escape_string()`

Return Values

Zero for success. Nonzero if an error occurred.

Example

```
MYSQL mysql;

mysql_init(&mysql);
if (!mysql_real_connect(&mysql,"host","user","passwd","database",0,NULL,0))
{
    fprintf(stderr, "Failed to connect to database: Error: %s\n",
          mysql_error(&mysql));
}

if (!mysql_set_character_set(&mysql, "utf8"))
{
    printf("New client character set: %s\n",
           mysql_character_set_name(&mysql));
}
```

7.74 mysql_set_local_infile_default()

```
void
mysql_set_local_infile_default(MYSQL *mysql);
```

Description

Sets the `LOAD DATA LOCAL` callback functions to the defaults used internally by the C client library. The library calls this function automatically if `mysql_set_local_infile_handler()` has not been called or does not supply valid functions for each of its callbacks.

Return Values

None.

Errors

None.

7.75 mysql_set_local_infile_handler()

```
void
mysql_set_local_infile_handler(MYSQL *mysql,
  int (*local_infile_init)(void **, const char *, void *),
  int (*local_infile_read)(void *, char *, unsigned int),
  void (*local_infile_end)(void *),
  int (*local_infile_error)(void *, char*, unsigned int),
  void *userdata);
```

Description

This function installs callbacks to be used during the execution of `LOAD DATA LOCAL` statements. It enables application programs to exert control over local (client-side) data file reading. The arguments are the connection handler, a set of pointers to callback functions, and a pointer to a data area that the callbacks can use to share information.

To use `mysql_set_local_infile_handler()`, you must write the following callback functions:

```
int
local_infile_init(void **ptr, const char *filename, void *userdata);
```

The initialization function. This is called once to do any setup necessary, open the data file, allocate data structures, and so forth. The first `void**` argument is a pointer to a pointer. You can set the pointer (that is, `*ptr`) to a value that will be passed to each of the other callbacks (as a `void*`). The callbacks can use this pointed-to value to maintain state information. The `userdata` argument is the same value that is passed to `mysql_set_local_infile_handler()`.

Make the initialization function return zero for success, nonzero for an error.

```
int
local_infile_read(void *ptr, char *buf, unsigned int buf_len);
```

The data-reading function. This is called repeatedly to read the data file. `buf` points to the buffer where the read data is stored, and `buf_len` is the maximum number of bytes that the callback can read and store in the buffer. (It can read fewer bytes, but should not read more.)

The return value is the number of bytes read, or zero when no more data could be read (this indicates EOF). Return a value less than zero if an error occurs.

```
void
local_infile_end(void *ptr)
```

The termination function. This is called once after `local_infile_read()` has returned zero (EOF) or an error. Within this function, deallocate any memory allocated by `local_infile_init()` and perform any other cleanup necessary. It is invoked even if the initialization function returns an error.

```
int
local_infile_error(void *ptr,
                   char *error_msg,
                   unsigned int error_msg_len);
```

The error-handling function. This is called to get a textual error message to return to the user in case any of your other functions returns an error. `error_msg` points to the buffer into which the message is written, and `error_msg_len` is the length of the buffer. Write the message as a null-terminated string, at most `error_msg_len`−1 bytes long.

The return value is the error number.

Typically, the other callbacks store the error message in the data structure pointed to by `ptr`, so that `local_infile_error()` can copy the message from there into `error_msg`.

After calling `mysql_set_local_infile_handler()` in your C code and passing pointers to your callback functions, you can then issue a `LOAD DATA LOCAL` statement (for example, by using `mysql_query()`). The client library automatically invokes your callbacks. The file name specified in `LOAD DATA LOCAL` will be passed as the second parameter to the `local_infile_init()` callback.

Return Values

None.

Errors

None.

7.76 mysql_set_server_option()

```
int
mysql_set_server_option(MYSQL *mysql, enum
                        enum_mysql_set_option option)
```

Description

Enables or disables an option for the connection. `option` can have one of the following values.

Option	Description
`MYSQL_OPTION_MULTI_STATEMENTS_ON`	Enable multiple-statement support
`MYSQL_OPTION_MULTI_STATEMENTS_OFF`	Disable multiple-statement support

If you enable multiple-statement support, you should retrieve results from calls to `mysql_query()` or `mysql_real_query()` by using a loop that calls `mysql_next_result()` to determine whether there are more results. For an example, see Chapter 23, *C API Multiple Statement Execution Support*.

Enabling multiple-statement support with `MYSQL_OPTION_MULTI_STATEMENTS_ON` does not have quite the same effect as enabling it by passing the `CLIENT_MULTI_STATEMENTS` flag to `mysql_real_connect()`: `CLIENT_MULTI_STATEMENTS` also enables `CLIENT_MULTI_RESULTS`. If you are using the `CALL` SQL statement in your programs, multiple-result support must be enabled; this means that `MYSQL_OPTION_MULTI_STATEMENTS_ON` by itself is insufficient to permit the use of `CALL`.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- CR_SERVER_LOST

  The connection to the server was lost during the query.

- ER_UNKNOWN_COM_ERROR

  The server did not support mysql_set_server_option() (which is the case that the server is older than 4.1.1) or the server did not support the option one tried to set.

7.77 mysql_shutdown()

```
int
mysql_shutdown(MYSQL *mysql,
               enum mysql_enum_shutdown_level shutdown_level)
```

Description

> **Note**
>
> mysql_shutdown() is deprecated and will be removed in a future version of MySQL. Instead, use mysql_query() to execute a SHUTDOWN statement.

Asks the database server to shut down. The connected user must have the SHUTDOWN privilege. MySQL servers support only one type of shutdown; shutdown_level must be equal to SHUTDOWN_DEFAULT. Dynamically linked executables that have been compiled with older versions of the libmysqlclient headers and call mysql_shutdown() must be used with the old libmysqlclient dynamic library.

An alternative to mysql_shutdown() is to use the SHUTDOWN SQL statement.

The shutdown process is described in The Server Shutdown Process.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- CR_COMMANDS_OUT_OF_SYNC

  Commands were executed in an improper order.

- CR_SERVER_GONE_ERROR

  The MySQL server has gone away.

- CR_SERVER_LOST

  The connection to the server was lost during the query.

- CR_UNKNOWN_ERROR

  An unknown error occurred.

7.78 mysql_sqlstate()

```
const char *
```

```
mysql_sqlstate(MYSQL *mysql)
```

Description

Returns a null-terminated string containing the SQLSTATE error code for the most recently executed SQL statement. The error code consists of five characters. `'00000'` means “no error.” The values are specified by ANSI SQL and ODBC. For a list of possible values, see Error Messages and Common Problems.

SQLSTATE values returned by `mysql_sqlstate()` differ from MySQL-specific error numbers returned by `mysql_errno()`. For example, the `mysql` client program displays errors using the following format, where `1146` is the `mysql_errno()` value and `'42S02'` is the corresponding `mysql_sqlstate()` value:

```
shell> SELECT * FROM no_such_table;
ERROR 1146 (42S02): Table 'test.no_such_table' doesn't exist
```

Not all MySQL error numbers are mapped to SQLSTATE error codes. The value `'HY000'` (general error) is used for unmapped error numbers.

If you call `mysql_sqlstate()` after `mysql_real_connect()` fails, `mysql_sqlstate()` might not return a useful value. For example, this happens if a host is blocked by the server and the connection is closed without any SQLSTATE value being sent to the client.

Return Values

A null-terminated character string containing the SQLSTATE error code.

See Also

See Section 7.15, “mysql_errno()”, Section 7.16, “mysql_error()”, and Section 11.27, “mysql_stmt_sqlstate()”.

7.79 mysql_ssl_set()

```
bool
mysql_ssl_set(MYSQL *mysql,
              const char *key,
              const char *cert,
              const char *ca,
              const char *capath,
              const char *cipher)
```

Description

`mysql_ssl_set()` is used for establishing encrypted connections using SSL. The `mysql` argument must be a valid connection handler. Any unused SSL arguments may be given as `NULL`.

If used, `mysql_ssl_set()` must be called before `mysql_real_connect()`. `mysql_ssl_set()` does nothing unless SSL support is enabled in the client library.

It is optional to call `mysql_ssl_set()` to obtain an encrypted connection because by default, MySQL programs attempt to connect using encryption if the server supports encrypted connections, falling back to an unencrypted connection if an encrypted connection cannot be established (see Configuring MySQL to Use Encrypted Connections). `mysql_ssl_set()` may be useful to applications that must specify particular certificate and key files, encryption ciphers, and so forth.

`mysql_ssl_set()` specifies SSL information such as certificate and key files for establishing an encrypted connection if such connections are available, but does not enforce any requirement that the

connection obtained be encrypted. To require an encrypted connection, use the technique described in Chapter 22, *C API Support for Encrypted Connections*.

For additional security relative to that provided by the default encryption, clients can supply a CA certificate matching the one used by the server and enable host name identity verification. In this way, the server and client place their trust in the same CA certificate and the client verifies that the host to which it connected is the one intended. For details, see Chapter 22, *C API Support for Encrypted Connections*.

`mysql_ssl_set()` is a convenience function that is essentially equivalent to this set of `mysql_options()` calls:

```
mysql_options(mysql, MYSQL_OPT_SSL_KEY,    key);
mysql_options(mysql, MYSQL_OPT_SSL_CERT,   cert);
mysql_options(mysql, MYSQL_OPT_SSL_CA,     ca);
mysql_options(mysql, MYSQL_OPT_SSL_CAPATH, capath);
mysql_options(mysql, MYSQL_OPT_SSL_CIPHER, cipher);
```

Because of that equivalence, applications can, instead of calling `mysql_ssl_set()`, call `mysql_options()` directly, omitting calls for those options for which the option value is `NULL`. Moreover, `mysql_options()` offers encrypted-connection options not available using `mysql_ssl_set()`, such as `MYSQL_OPT_SSL_MODE` to specify the security state of the connection, and `MYSQL_OPT_TLS_VERSION` to specify the protocols the client permits for encrypted connections.

Arguments:

- `mysql`: The connection handler returned from `mysql_init()`.
- `key`: The path name of the client private key file.
- `cert`: The path name of the client public key certificate file.
- `ca`: The path name of the Certificate Authority (CA) certificate file. This option, if used, must specify the same certificate used by the server.
- `capath`: The path name of the directory that contains trusted SSL CA certificate files.
- `cipher`: The list of permissible ciphers for SSL encryption.

Return Values

This function always returns `0`. If SSL setup is incorrect, a subsequent `mysql_real_connect()` call returns an error when you attempt to connect.

7.80 mysql_stat()

```
const char *
mysql_stat(MYSQL *mysql)
```

Description

Returns a character string containing information similar to that provided by the `mysqladmin status` command. This includes uptime in seconds and the number of running threads, questions, reloads, and open tables.

Return Values

A character string describing the server status. `NULL` if an error occurred.

Errors

- CR_COMMANDS_OUT_OF_SYNC

  Commands were executed in an improper order.

- CR_SERVER_GONE_ERROR

  The MySQL server has gone away.

- CR_SERVER_LOST

  The connection to the server was lost during the query.

- CR_UNKNOWN_ERROR

  An unknown error occurred.

7.81 mysql_store_result()

```
MYSQL_RES *
mysql_store_result(MYSQL *mysql)
```

Description

> **Note**
>
> mysql_store_result() is a synchronous function. Its asynchronous counterpart is mysql_store_result_nonblocking(), for use by applications that require asynchronous communication with the server. See Chapter 12, *C API Asynchronous Interface*.

After invoking mysql_query() or mysql_real_query(), you must call mysql_store_result() or mysql_use_result() for every statement that successfully produces a result set (SELECT, SHOW, DESCRIBE, EXPLAIN, CHECK TABLE, and so forth). You must also call mysql_free_result() after you are done with the result set.

You need not call mysql_store_result() or mysql_use_result() for other statements, but it does not do any harm or cause any notable performance degradation if you call mysql_store_result() in all cases. You can detect whether the statement has a result set by checking whether mysql_store_result() returns a nonzero value (more about this later).

If you enable multiple-statement support, you should retrieve results from calls to mysql_query() or mysql_real_query() by using a loop that calls mysql_next_result() to determine whether there are more results. For an example, see Chapter 23, *C API Multiple Statement Execution Support*.

To determine whether a statement returns a result set, call mysql_field_count(). See Section 7.23, “mysql_field_count()”.

mysql_store_result() reads the entire result of a query to the client, allocates a MYSQL_RES structure, and places the result into this structure.

mysql_store_result() returns NULL if the statement did not return a result set (for example, if it was an INSERT statement), or an error occurred and reading of the result set failed.

An empty result set is returned if there are no rows returned. (An empty result set differs from a null pointer as a return value.)

After you have called `mysql_store_result()` and gotten back a result that is not a null pointer, you can call `mysql_num_rows()` to find out how many rows are in the result set.

You can call `mysql_fetch_row()` to fetch rows from the result set, or `mysql_row_seek()` and `mysql_row_tell()` to obtain or set the current row position within the result set.

See Section 29.1, "Why mysql_store_result() Sometimes Returns NULL After mysql_query() Returns Success".

Return Values

A pointer to a `MYSQL_RES` result structure with the results. `NULL` if the statement did not return a result set or an error occurred. To determine whether an error occurred, check whether `mysql_error()` returns a nonempty string, `mysql_errno()` returns nonzero, or `mysql_field_count()` returns zero.

Errors

`mysql_store_result()` resets `mysql_error()` and `mysql_errno()` if it succeeds.

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_OUT_OF_MEMORY`

  Out of memory.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

7.82 mysql_thread_id()

```
unsigned long
mysql_thread_id(MYSQL *mysql)
```

Description

Returns the thread ID of the current connection. This value can be used as an argument to `mysql_kill()` to kill the thread.

If the connection is lost and you reconnect with `mysql_ping()`, the thread ID changes. This means you should not get the thread ID and store it for later. You should get it when you need it.

> **Note**
>
> This function does not work correctly if thread IDs become larger than 32 bits, which can occur on some systems. To avoid problems with `mysql_thread_id()`, do not

> use it. To get the connection ID, execute a `SELECT CONNECTION_ID()` query and retrieve the result.

Return Values

The thread ID of the current connection.

Errors

None.

7.83 mysql_use_result()

```
MYSQL_RES *
mysql_use_result(MYSQL *mysql)
```

Description

After invoking `mysql_query()` or `mysql_real_query()`, you must call `mysql_store_result()` or `mysql_use_result()` for every statement that successfully produces a result set (`SELECT`, `SHOW`, `DESCRIBE`, `EXPLAIN`, `CHECK TABLE`, and so forth). You must also call `mysql_free_result()` after you are done with the result set.

`mysql_use_result()` initiates a result set retrieval but does not actually read the result set into the client like `mysql_store_result()` does. Instead, each row must be retrieved individually by making calls to `mysql_fetch_row()`. This reads the result of a query directly from the server without storing it in a temporary table or local buffer, which is somewhat faster and uses much less memory than `mysql_store_result()`. The client allocates memory only for the current row and a communication buffer that may grow up to `max_allowed_packet` bytes.

On the other hand, you should not use `mysql_use_result()` for locking reads if you are doing a lot of processing for each row on the client side, or if the output is sent to a screen on which the user may type a `^S` (stop scroll). This ties up the server and prevent other threads from updating any tables from which the data is being fetched.

When using `mysql_use_result()`, you must execute `mysql_fetch_row()` until a `NULL` value is returned, otherwise, the unfetched rows are returned as part of the result set for your next query. The C API gives the error `Commands out of sync; you can't run this command now` if you forget to do this!

You may not use `mysql_data_seek()`, `mysql_row_seek()`, `mysql_row_tell()`, `mysql_num_rows()`, or `mysql_affected_rows()` with a result returned from `mysql_use_result()`, nor may you issue other queries until `mysql_use_result()` has finished. (However, after you have fetched all the rows, `mysql_num_rows()` accurately returns the number of rows fetched.)

You must call `mysql_free_result()` once you are done with the result set.

Return Values

A `MYSQL_RES` result structure. `NULL` if an error occurred.

Errors

`mysql_use_result()` resets `mysql_error()` and `mysql_errno()` if it succeeds.

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_OUT_OF_MEMORY`

  Out of memory.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

7.84 mysql_warning_count()

```
unsigned int
mysql_warning_count(MYSQL *mysql)
```

Description

Returns the number of errors, warnings, and notes generated during execution of the previous SQL statement.

Return Values

The warning count.

Errors

None.

Chapter 8 C API Prepared Statements

The MySQL client/server protocol provides for the use of prepared statements. This capability uses the `MYSQL_STMT` statement handler data structure returned by the `mysql_stmt_init()` initialization function. Prepared execution is an efficient way to execute a statement more than once. The statement is first parsed to prepare it for execution. Then it is executed one or more times at a later time, using the statement handler returned by the initialization function.

Prepared execution is faster than direct execution for statements executed more than once, primarily because the query is parsed only once. In the case of direct execution, the query is parsed every time it is executed. Prepared execution also can provide a reduction of network traffic because for each execution of the prepared statement, it is necessary only to send the data for the parameters.

Prepared statements might not provide a performance increase in some situations. For best results, test your application both with prepared and nonprepared statements and choose whichever yields best performance.

Another advantage of prepared statements is that it uses a binary protocol that makes data transfer between client and server more efficient.

For a list of SQL statements that can be used as prepared statements, see Prepared Statements.

Metadata changes to tables or views referred to by prepared statements are detected and cause automatic repreparation of the statement when it is next executed. For more information, see Caching of Prepared Statements and Stored Programs.

Chapter 9 C API Prepared Statement Data Structures

Table of Contents

Prepared statements use several data structures:

- To obtain a statement handler, pass a `MYSQL` connection handler to `mysql_stmt_init()`, which returns a pointer to a `MYSQL_STMT` data structure. This structure is used for further operations with the statement. To specify the statement to prepare, pass the `MYSQL_STMT` pointer and the statement string to `mysql_stmt_prepare()`.
- To provide input parameters for a prepared statement, set up `MYSQL_BIND` structures and pass them to `mysql_stmt_bind_param()`. To receive output column values, set up `MYSQL_BIND` structures and pass them to `mysql_stmt_bind_result()`.

  `MYSQL_BIND` structures are also used with `mysql_bind_param()`, which enables defining attributes that apply to the next query sent to the server.
- The `MYSQL_TIME` structure is used to transfer temporal data in both directions.

The following discussion describes the prepared statement data types in detail. For examples that show how to use them, see Section 11.10, “mysql_stmt_execute()”, and Section 11.11, “mysql_stmt_fetch()”.

- `MYSQL_STMT`

  This structure is a handler for a prepared statement. A handler is created by calling `mysql_stmt_init()`, which returns a pointer to a `MYSQL_STMT`. The handler is used for all subsequent operations with the statement until you close it with `mysql_stmt_close()`, at which point the handler becomes invalid and should no longer be used.

  The `MYSQL_STMT` structure has no members intended for application use. Applications should not try to copy a `MYSQL_STMT` structure. There is no guarantee that such a copy will be usable.

  Multiple statement handlers can be associated with a single connection. The limit on the number of handlers depends on the available system resources.
- `MYSQL_BIND`

  This structure is used both for statement input (data values sent to the server) and output (result values returned from the server):

  - For input, use `MYSQL_BIND` structures with `mysql_bind_param()` to define attributes for a query. (In the following discussion, treat any mention of statement parameters for prepared statements as also applying to query attributes.)
  - For output, use `MYSQL_BIND` structures with `mysql_stmt_bind_result()` to bind buffers to result set columns, for use in fetching rows with `mysql_stmt_fetch()`.

  To use a `MYSQL_BIND` structure, zero its contents to initialize it, then set its members appropriately. For example, to declare and initialize an array of three `MYSQL_BIND` structures, use this code:

```
MYSQL_BIND bind[3];
memset(bind, 0, sizeof(bind));
```

The `MYSQL_BIND` structure contains the following members for use by application programs. For several of the members, the manner of use depends on whether the structure is used for input or output.

- `enum enum_field_types buffer_type`

  The type of the buffer. This member indicates the data type of the C language variable bound to a statement parameter or result set column. For input, `buffer_type` indicates the type of the variable containing the value to be sent to the server. For output, it indicates the type of the variable into which a value received from the server should be stored. For permissible `buffer_type` values, see Section 9.1, “C API Prepared Statement Type Codes”.

- `void *buffer`

  A pointer to the buffer to be used for data transfer. This is the address of a C language variable.

  For input, `buffer` is a pointer to the variable in which you store the data value for a statement parameter. When you call `mysql_stmt_execute()`, MySQL use the value stored in the variable in place of the corresponding parameter marker in the statement (specified with `?` in the statement string).

  For output, `buffer` is a pointer to the variable in which to return a result set column value. When you call `mysql_stmt_fetch()`, MySQL stores a column value from the current row of the result set in this variable. You can access the value when the call returns.

  To minimize the need for MySQL to perform type conversions between C language values on the client side and SQL values on the server side, use C variables that have types similar to those of the corresponding SQL values:

  - For numeric data types, `buffer` should point to a variable of the proper numeric C type. For integer variables (which can be `char` for single-byte values or an integer type for larger values), you should also indicate whether the variable has the `unsigned` attribute by setting the `is_unsigned` member, described later.
  - For character (nonbinary) and binary string data types, `buffer` should point to a character buffer.
  - For date and time data types, `buffer` should point to a `MYSQL_TIME` structure.

  For guidelines about mapping between C types and SQL types and notes about type conversions, see Section 9.1, “C API Prepared Statement Type Codes”, and Section 9.2, “C API Prepared Statement Type Conversions”.

- `unsigned long buffer_length`

  The actual size of `*buffer` in bytes. This indicates the maximum amount of data that can be stored in the buffer. For character and binary C data, the `buffer_length` value specifies the length of `*buffer` when used with `mysql_stmt_bind_param()` to specify input values, or

the maximum number of output data bytes that can be fetched into the buffer when used with `mysql_stmt_bind_result()`.

- `unsigned long *length`

  A pointer to an `unsigned long` variable that indicates the actual number of bytes of data stored in `*buffer`. `length` is used for character or binary C data.

  For input parameter data binding, set `*length` to indicate the actual length of the parameter value stored in `*buffer`. This is used by `mysql_stmt_execute()`.

  For output value binding, MySQL sets `*length` when you call `mysql_stmt_fetch()`. The `mysql_stmt_fetch()` return value determines how to interpret the length:

  - If the return value is 0, `*length` indicates the actual length of the parameter value.
  - If the return value is `MYSQL_DATA_TRUNCATED`, `*length` indicates the nontruncated length of the parameter value. In this case, the minimum of `*length` and `buffer_length` indicates the actual length of the value.

  `length` is ignored for numeric and temporal data types because the `buffer_type` value determines the length of the data value.

  If you must determine the length of a returned value before fetching it, see Section 11.11, “mysql_stmt_fetch()”, for some strategies.

- `bool *is_null`

  This member points to a `bool` variable that is true if a value is `NULL`, false if it is not `NULL`. For input, set `*is_null` to true to indicate that you are passing a `NULL` value as a statement parameter.

  `is_null` is a *pointer* to a boolean scalar, not a boolean scalar, to provide flexibility in how you specify `NULL` values:

  - If your data values are always `NULL`, use `MYSQL_TYPE_NULL` as the `buffer_type` value when you bind the column. The other `MYSQL_BIND` members, including `is_null`, do not matter.
  - If your data values are always `NOT NULL`, set `is_null = (bool*) 0`, and set the other members appropriately for the variable you are binding.
  - In all other cases, set the other members appropriately and set `is_null` to the address of a `bool` variable. Set that variable's value to true or false appropriately between executions to indicate whether the corresponding data value is `NULL` or `NOT NULL`, respectively.

  For output, when you fetch a row, MySQL sets the value pointed to by `is_null` to true or false according to whether the result set column value returned from the statement is or is not `NULL`.

- `bool is_unsigned`

  This member applies for C variables with data types that can be `unsigned` (`char`, `short int`, `int`, `long long int`). Set `is_unsigned` to true if the variable pointed to by `buffer` is `unsigned` and false otherwise. For example, if you bind a `signed char` variable to `buffer`, specify a type code of `MYSQL_TYPE_TINY` and set `is_unsigned` to false. If you bind an `unsigned char` instead, the type

code is the same but `is_unsigned` should be true. (For `char`, it is not defined whether it is signed or unsigned, so it is best to be explicit about signedness by using `signed char` or `unsigned char`.)

`is_unsigned` applies only to the C language variable on the client side. It indicates nothing about the signedness of the corresponding SQL value on the server side. For example, if you use an `int` variable to supply a value for a `BIGINT UNSIGNED` column, `is_unsigned` should be false because `int` is a signed type. If you use an `unsigned int` variable to supply a value for a `BIGINT` column, `is_unsigned` should be true because `unsigned int` is an unsigned type. MySQL performs the proper conversion between signed and unsigned values in both directions, although a warning occurs if truncation results.

- `bool *error`

  For output, set this member to point to a `bool` variable to have truncation information for the parameter stored there after a row fetching operation. When truncation reporting is enabled, `mysql_stmt_fetch()` returns `MYSQL_DATA_TRUNCATED` and `*error` is true in the `MYSQL_BIND` structures for parameters in which truncation occurred. Truncation indicates loss of sign or significant digits, or that a string was too long to fit in a column. Truncation reporting is enabled by default, but can be controlled by calling `mysql_options()` with the `MYSQL_REPORT_DATA_TRUNCATION` option.

- `MYSQL_TIME`

  This structure is used to send and receive `DATE`, `TIME`, `DATETIME`, and `TIMESTAMP` data directly to and from the server. Set the `buffer` member to point to a `MYSQL_TIME` structure, and set the `buffer_type` member of a `MYSQL_BIND` structure to one of the temporal types (`MYSQL_TYPE_TIME`, `MYSQL_TYPE_DATE`, `MYSQL_TYPE_DATETIME`, `MYSQL_TYPE_TIMESTAMP`).

  The `MYSQL_TIME` structure contains the members listed in the following table.

Member	Description
`unsigned int year`	The year
`unsigned int month`	The month of the year
`unsigned int day`	The day of the month
`unsigned int hour`	The hour of the day
`unsigned int minute`	The minute of the hour
`unsigned int second`	The second of the minute
`bool neg`	A boolean flag indicating whether the time is negative
`unsigned long second_part`	The fractional part of the second in microseconds

  Only those parts of a `MYSQL_TIME` structure that apply to a given type of temporal value are used. The `year`, `month`, and `day` elements are used for `DATE`, `DATETIME`, and `TIMESTAMP` values. The `hour`, `minute`, and `second` elements are used for `TIME`, `DATETIME`, and `TIMESTAMP` values. See Chapter 24, *C API Prepared Statement Handling of Date and Time Values*.

9.1 C API Prepared Statement Type Codes

The `buffer_type` member of `MYSQL_BIND` structures indicates the data type of the C language variable bound to a statement parameter or result set column. For input, `buffer_type` indicates the type of the variable containing the value to be sent to the server. For output, it indicates the type of the variable into which a value received from the server should be stored.

The following table shows the permissible values for the `buffer_type` member of `MYSQL_BIND` structures for input values sent to the server. The table shows the C variable types that you can use, the corresponding type codes, and the SQL data types for which the supplied value can be used without conversion. Choose the `buffer_type` value according to the data type of the C language variable that you are binding. For the integer types, you should also set the `is_unsigned` member to indicate whether the variable is signed or unsigned.

Table 9.1 Permissible Input Data Types for MYSQL_BIND Structures

Input Variable C Type	`buffer_type` Value	SQL Type of Destination Value
`signed char`	`MYSQL_TYPE_TINY`	`TINYINT`
`short int`	`MYSQL_TYPE_SHORT`	`SMALLINT`
`int`	`MYSQL_TYPE_LONG`	`INT`
`long long int`	`MYSQL_TYPE_LONGLONG`	`BIGINT`
`float`	`MYSQL_TYPE_FLOAT`	`FLOAT`
`double`	`MYSQL_TYPE_DOUBLE`	`DOUBLE`
`MYSQL_TIME`	`MYSQL_TYPE_TIME`	`TIME`
`MYSQL_TIME`	`MYSQL_TYPE_DATE`	`DATE`
`MYSQL_TIME`	`MYSQL_TYPE_DATETIME`	`DATETIME`
`MYSQL_TIME`	`MYSQL_TYPE_TIMESTAMP`	`TIMESTAMP`
`char[]`	`MYSQL_TYPE_STRING`	`TEXT`, `CHAR`, `VARCHAR`
`char[]`	`MYSQL_TYPE_BLOB`	`BLOB`, `BINARY`, `VARBINARY`
	`MYSQL_TYPE_NULL`	`NULL`

Use `MYSQL_TYPE_NULL` as indicated in the description for the `is_null` member in Chapter 9, *C API Prepared Statement Data Structures*.

For input string data, use `MYSQL_TYPE_STRING` or `MYSQL_TYPE_BLOB` depending on whether the value is a character (nonbinary) or binary string:

- `MYSQL_TYPE_STRING` indicates character input string data. The value is assumed to be in the character set indicated by the `character_set_client` system variable. If the server stores the value into a column with a different character set, it converts the value to that character set.
- `MYSQL_TYPE_BLOB` indicates binary input string data. The value is treated as having the `binary` character set. That is, it is treated as a byte string and no conversion occurs.

The following table shows the permissible values for the `buffer_type` member of `MYSQL_BIND` structures for output values received from the server. The table shows the SQL types of received values, the corresponding type codes that such values have in result set metadata, and the recommended C language data types to bind to the `MYSQL_BIND` structure to receive the SQL values without conversion. Choose the `buffer_type` value according to the data type of the C language variable that you are binding. For the integer types, you should also set the `is_unsigned` member to indicate whether the variable is signed or unsigned.

Table 9.2 Permissible Output Data Types for MYSQL_BIND Structures

SQL Type of Received Value	`buffer_type` Value	Output Variable C Type
`TINYINT`	`MYSQL_TYPE_TINY`	`signed char`

SQL Type of Received Value	buffer_type Value	Output Variable C Type
SMALLINT	MYSQL_TYPE_SHORT	short int
MEDIUMINT	MYSQL_TYPE_INT24	int
INT	MYSQL_TYPE_LONG	int
BIGINT	MYSQL_TYPE_LONGLONG	long long int
FLOAT	MYSQL_TYPE_FLOAT	float
DOUBLE	MYSQL_TYPE_DOUBLE	double
DECIMAL	MYSQL_TYPE_NEWDECIMAL	char[]
YEAR	MYSQL_TYPE_SHORT	short int
TIME	MYSQL_TYPE_TIME	MYSQL_TIME
DATE	MYSQL_TYPE_DATE	MYSQL_TIME
DATETIME	MYSQL_TYPE_DATETIME	MYSQL_TIME
TIMESTAMP	MYSQL_TYPE_TIMESTAMP	MYSQL_TIME
CHAR, BINARY	MYSQL_TYPE_STRING	char[]
VARCHAR, VARBINARY	MYSQL_TYPE_VAR_STRING	char[]
TINYBLOB, TINYTEXT	MYSQL_TYPE_TINY_BLOB	char[]
BLOB, TEXT	MYSQL_TYPE_BLOB	char[]
MEDIUMBLOB, MEDIUMTEXT	MYSQL_TYPE_MEDIUM_BLOB	char[]
LONGBLOB, LONGTEXT	MYSQL_TYPE_LONG_BLOB	char[]
BIT	MYSQL_TYPE_BIT	char[]

9.2 C API Prepared Statement Type Conversions

Prepared statements transmit data between the client and server using C language variables on the client side that correspond to SQL values on the server side. If there is a mismatch between the C variable type on the client side and the corresponding SQL value type on the server side, MySQL performs implicit type conversions in both directions.

MySQL knows the type code for the SQL value on the server side. The `buffer_type` value in the `MYSQL_BIND` structure indicates the type code of the C variable that holds the value on the client side. The two codes together tell MySQL what conversion must be performed, if any. Here are some examples:

- If you use `MYSQL_TYPE_LONG` with an `int` variable to pass an integer value to the server that is to be stored into a `FLOAT` column, MySQL converts the value to floating-point format before storing it.
- If you fetch an SQL `MEDIUMINT` column value, but specify a `buffer_type` value of `MYSQL_TYPE_LONGLONG` and use a C variable of type `long long int` as the destination buffer, MySQL converts the `MEDIUMINT` value (which requires less than 8 bytes) for storage into the `long long int` (an 8-byte variable).
- If you fetch a numeric column with a value of 255 into a `char[4]` character array and specify a `buffer_type` value of `MYSQL_TYPE_STRING`, the resulting value in the array is a 4-byte string `'255\0'`.
- MySQL returns `DECIMAL` values as the string representation of the original server-side value, which is why the corresponding C type is `char[]`. For example, `12.345` is returned to the client as `'12.345'`. If you specify `MYSQL_TYPE_NEWDECIMAL` and bind a string buffer to the `MYSQL_BIND` structure,

mysql_stmt_fetch() stores the value in the buffer as a string without conversion. If instead you specify a numeric variable and type code, mysql_stmt_fetch() converts the string-format DECIMAL value to numeric form.

- For the MYSQL_TYPE_BIT type code, BIT values are returned into a string buffer, which is why the corresponding C type is char[]. The value represents a bit string that requires interpretation on the client side. To return the value as a type that is easier to deal with, you can cause the value to be cast to integer using either of the following types of expressions:

  ```
  SELECT bit_col + 0 FROM t
  SELECT CAST(bit_col AS UNSIGNED) FROM t
  ```

  To retrieve the value, bind an integer variable large enough to hold the value and specify the appropriate corresponding integer type code.

Before binding variables to the MYSQL_BIND structures that are to be used for fetching column values, you can check the type codes for each column of the result set. This might be desirable if you want to determine which variable types would be best to use to avoid type conversions. To get the type codes, call mysql_stmt_result_metadata() after preparing the statement with mysql_stmt_prepare(). The metadata provides access to the type codes for the result set as described in Section 11.23, “mysql_stmt_result_metadata()”, and Chapter 5, *C API Data Structures*.

To determine whether output string values in a result set returned from the server contain binary or nonbinary data, check whether the charsetnr value of the result set metadata is 63 (see Chapter 5, *C API Data Structures*). If so, the character set is binary, which indicates binary rather than nonbinary data. This enables you to distinguish BINARY from CHAR, VARBINARY from VARCHAR, and the BLOB types from the TEXT types.

If you cause the max_length member of the MYSQL_FIELD column metadata structures to be set (by calling mysql_stmt_attr_set()), be aware that the max_length values for the result set indicate the lengths of the longest string representation of the result values, not the lengths of the binary representation. That is, max_length does not necessarily correspond to the size of the buffers needed to fetch the values with the binary protocol used for prepared statements. Choose the size of the buffers according to the types of the variables into which you fetch the values. For example, a TINYINT column containing the value -128 might have a max_length value of 4. But the binary representation of any TINYINT value requires only 1 byte for storage, so you can supply a signed char variable in which to store the value and set is_unsigned to indicate that values are signed.

Metadata changes to tables or views referred to by prepared statements are detected and cause automatic repreparation of the statement when it is next executed. For more information, see Caching of Prepared Statements and Stored Programs.

Chapter 10 C API Prepared Statement Function Overview

The following list summarizes the functions available for prepared statement processing. For greater detail, see the descriptions in Chapter 11, *C API Prepared Statement Function Descriptions*.

- `mysql_stmt_affected_rows()`: Returns the number of rows changed, deleted, or inserted by prepared `UPDATE`, `DELETE`, or `INSERT` statement.
- `mysql_stmt_attr_get()`: Gets the value of an attribute for a prepared statement.
- `mysql_stmt_attr_set()`: Sets an attribute for a prepared statement.
- `mysql_stmt_bind_param()`: Associates application data buffers with the parameter markers in a prepared SQL statement.
- `mysql_stmt_bind_result()`: Associates application data buffers with columns in a result set.
- `mysql_stmt_close()`: Frees memory used by a prepared statement.
- `mysql_stmt_data_seek()`: Seeks to an arbitrary row number in a statement result set.
- `mysql_stmt_errno()`: Returns the error number for the last statement execution.
- `mysql_stmt_error()`: Returns the error message for the last statement execution.
- `mysql_stmt_execute()`: Executes a prepared statement.
- `mysql_stmt_fetch()`: Fetches the next row of data from a result set and returns data for all bound columns.
- `mysql_stmt_fetch_column()`: Fetches data for one column of the current row of a result set.
- `mysql_stmt_field_count()`: Returns the number of result columns for the most recent statement.
- `mysql_stmt_free_result()`: Frees the resources allocated to a statement handler.
- `mysql_stmt_init()`: Allocates memory for a `MYSQL_STMT` structure and initializes it.
- `mysql_stmt_insert_id()`: Returns the ID generated for an `AUTO_INCREMENT` column by a prepared statement.
- `mysql_stmt_next_result()`: Returns/initiates the next result in a multiple-result execution.
- `mysql_stmt_num_rows()`: Returns the row count from a buffered statement result set.
- `mysql_stmt_param_count()`: Returns the number of parameters in a prepared statement.
- `mysql_stmt_param_metadata()`: Returns parameter metadata in the form of a result set. (This function actually does nothing.)
- `mysql_stmt_prepare()`: Prepares an SQL statement string for execution.
- `mysql_stmt_reset()`: Resets the statement buffers in the server.
- `mysql_stmt_result_metadata()`: Returns prepared statement metadata in the form of a result set.
- `mysql_stmt_row_seek()`: Seeks to a row offset in a statement result set, using value returned from `mysql_stmt_row_tell()`.

- `mysql_stmt_row_tell()`: Returns the statement row cursor position.
- `mysql_stmt_send_long_data()`: Sends long data in chunks to server.
- `mysql_stmt_sqlstate()`: Returns the SQLSTATE error code for the last statement execution.
- `mysql_stmt_store_result()`: Retrieves a complete result set to the client.

Call `mysql_stmt_init()` to create a statement handler, then `mysql_stmt_prepare()` to prepare the statement string, `mysql_stmt_bind_param()` to supply the parameter data, and `mysql_stmt_execute()` to execute the statement. You can repeat the `mysql_stmt_execute()` by changing parameter values in the respective buffers supplied through `mysql_stmt_bind_param()`.

You can send text or binary data in chunks to server using `mysql_stmt_send_long_data()`. See Section 11.26, “mysql_stmt_send_long_data()”.

If the statement is a `SELECT` or any other statement that produces a result set, `mysql_stmt_prepare()` also returns the result set metadata information in the form of a `MYSQL_RES` result set through `mysql_stmt_result_metadata()`.

You can supply the result buffers using `mysql_stmt_bind_result()`, so that the `mysql_stmt_fetch()` automatically returns data to these buffers. This is row-by-row fetching.

When statement execution has been completed, close the statement handler using `mysql_stmt_close()` so that all resources associated with it can be freed. At that point the handler becomes invalid and should no longer be used.

If you obtained a `SELECT` statement's result set metadata by calling `mysql_stmt_result_metadata()`, you should also free the metadata using `mysql_free_result()`.

Execution Steps

To prepare and execute a statement, an application follows these steps:

1. Create a prepared statement handler with `mysql_stmt_init()`. To prepare the statement on the server, call `mysql_stmt_prepare()` and pass it a string containing the SQL statement.
2. Set the values of any parameters using `mysql_stmt_bind_param()`. All parameters must be set. Otherwise, statement execution returns an error or produces unexpected results.
3. Call `mysql_stmt_execute()` to execute the statement.
4. If the statement will produce a result set, call `mysql_stmt_result_metadata()` if it is desired to obtain the result set metadata. This metadata is itself in the form of result set, albeit a separate one from the one that contains the rows returned by the query. The metadata result set indicates the number of columns in the result and contains information about each one.
5. If the statement produces a result set, bind the data buffers to use for retrieving the row values by calling `mysql_stmt_bind_result()`.
6. Fetch the data into the buffers row by row by calling `mysql_stmt_fetch()` repeatedly until no more rows are found.
7. Repeat steps 3 through 6 as necessary, by changing the parameter values and re-executing the statement.

When `mysql_stmt_prepare()` is called, the MySQL client/server protocol performs these actions:

- The server parses the statement and sends the okay status back to the client by assigning a statement ID. It also sends total number of parameters, a column count, and its metadata if it is a result set oriented statement. All syntax and semantics of the statement are checked by the server during this call.
- The client uses this statement ID for the further operations, so that the server can identify the statement from among its pool of statements.

When `mysql_stmt_execute()` is called, the MySQL client/server protocol performs these actions:

- The client uses the statement handler and sends the parameter data to the server.
- The server identifies the statement using the ID provided by the client, replaces the parameter markers with the newly supplied data, and executes the statement. If the statement produces a result set, the server sends the data back to the client. Otherwise, it sends an okay status and the number of rows changed, deleted, or inserted.

When `mysql_stmt_fetch()` is called, the MySQL client/server protocol performs these actions:

- The client reads the data from the current row of the result set and places it into the application data buffers by doing the necessary conversions. If the application buffer type is same as that of the field type returned from the server, the conversions are straightforward.

If an error occurs, you can get the statement error number, error message, and SQLSTATE code using `mysql_stmt_errno()`, `mysql_stmt_error()`, and `mysql_stmt_sqlstate()`, respectively.

Prepared Statement Logging

For prepared statements that are executed with the `mysql_stmt_prepare()` and `mysql_stmt_execute()` C API functions, the server writes `Prepare` and `Execute` lines to the general query log so that you can tell when statements are prepared and executed.

Suppose that you prepare and execute a statement as follows:

1. Call `mysql_stmt_prepare()` to prepare the statement string `"SELECT ?"`.
2. Call `mysql_stmt_bind_param()` to bind the value `3` to the parameter in the prepared statement.
3. Call `mysql_stmt_execute()` to execute the prepared statement.

As a result of the preceding calls, the server writes the following lines to the general query log:

```
Prepare  [1] SELECT ?
Execute  [1] SELECT 3
```

Each `Prepare` and `Execute` line in the log is tagged with a `[N]` statement identifier so that you can keep track of which prepared statement is being logged. *N* is a positive integer. If there are multiple prepared statements active simultaneously for the client, *N* may be greater than 1. Each `Execute` lines shows a prepared statement after substitution of data values for `?` parameters.

Chapter 11 C API Prepared Statement Function Descriptions

Table of Contents

To prepare and execute queries, use the functions described in detail in the following sections.

All functions that operate with a `MYSQL_STMT` structure begin with the prefix `mysql_stmt_`.

To create a `MYSQL_STMT` handler, use the `mysql_stmt_init()` function.

11.1 mysql_stmt_affected_rows()

```
uint64_t
mysql_stmt_affected_rows(MYSQL_STMT *stmt)
```

Description

`mysql_stmt_affected_rows()` may be called immediately after executing a statement with `mysql_stmt_execute()`. It is like `mysql_affected_rows()` but for prepared statements. For a description of what the affected-rows value returned by this function means, See Section 7.1, “mysql_affected_rows()”.

Errors

None.

Example

See the Example in Section 11.10, “mysql_stmt_execute()”.

11.2 mysql_stmt_attr_get()

```
bool
mysql_stmt_attr_get(MYSQL_STMT *stmt,
                    enum enum_stmt_attr_type option,
                    void *arg)
```

Description

Can be used to get the current value for a statement attribute.

The `option` argument is the option that you want to get; the `arg` should point to a variable that should contain the option value. If the option is an integer, `arg` should point to the value of the integer.

See Section 11.3, “mysql_stmt_attr_set()”, for a list of options and option types.

Return Values

Zero for success. Nonzero if `option` is unknown.

Errors

None.

11.3 mysql_stmt_attr_set()

```
bool
mysql_stmt_attr_set(MYSQL_STMT *stmt,
                    enum enum_stmt_attr_type option,
                    const void *arg)
```

Description

Can be used to affect behavior for a prepared statement. This function may be called multiple times to set several options.

The `option` argument is the option that you want to set. The `arg` argument is the value for the option. `arg` should point to a variable that is set to the desired attribute value. The variable type is as indicated in the following table.

The following table shows the possible `option` values.

Option	Argument Type	Function
`STMT_ATTR_UPDATE_MAX_LENGTH`	`bool *`	If set to 1, causes `mysql_stmt_store_result()` to update the metadata `MYSQL_FIELD->max_length` value.
`STMT_ATTR_CURSOR_TYPE`	`unsigned long *`	Type of cursor to open for statement when `mysql_stmt_execute()` is invoked. `*arg` can be `CURSOR_TYPE_NO_CURSOR`

Option	Argument Type	Function
		(the default) or `CURSOR_TYPE_READ_ONLY`.
`STMT_ATTR_PREFETCH_ROWS`	`unsigned long *`	Number of rows to fetch from server at a time when using a cursor. `*arg` can be in the range from 1 to the maximum value of `unsigned long`. The default is 1.

If you use the `STMT_ATTR_CURSOR_TYPE` option with `CURSOR_TYPE_READ_ONLY`, a cursor is opened for the statement when you invoke `mysql_stmt_execute()`. If there is already an open cursor from a previous `mysql_stmt_execute()` call, it closes the cursor before opening a new one. `mysql_stmt_reset()` also closes any open cursor before preparing the statement for re-execution. `mysql_stmt_free_result()` closes any open cursor.

If you open a cursor for a prepared statement, `mysql_stmt_store_result()` is unnecessary, because that function causes the result set to be buffered on the client side.

Return Values

Zero for success. Nonzero if `option` is unknown.

Errors

None.

Example

The following example opens a cursor for a prepared statement and sets the number of rows to fetch at a time to 5:

```
MYSQL_STMT *stmt;
int rc;
unsigned long type;
unsigned long prefetch_rows = 5;

stmt = mysql_stmt_init(mysql);
type = (unsigned long) CURSOR_TYPE_READ_ONLY;
rc = mysql_stmt_attr_set(stmt, STMT_ATTR_CURSOR_TYPE, (void*) &type);
/* ... check return value ... */
rc = mysql_stmt_attr_set(stmt, STMT_ATTR_PREFETCH_ROWS,
                         (void*) &prefetch_rows);
/* ... check return value ... */
```

11.4 mysql_stmt_bind_param()

```
bool
mysql_stmt_bind_param(MYSQL_STMT *stmt,
                      MYSQL_BIND *bind)
```

Description

`mysql_stmt_bind_param()` is used to bind input data for the parameter markers in the SQL statement that was passed to `mysql_stmt_prepare()`. It uses `MYSQL_BIND` structures to supply the data. `bind` is the address of an array of `MYSQL_BIND` structures. The client library expects the array to contain one element for each `?` parameter marker that is present in the query.

Suppose that you prepare the following statement:

```
INSERT INTO mytbl VALUES(?,?,?)
```

When you bind the parameters, the array of `MYSQL_BIND` structures must contain three elements, and can be declared like this:

```
MYSQL_BIND bind[3];
```

For a description of the members of the `MYSQL_BIND` structure and how they should be set to provide input values, see Chapter 9, *C API Prepared Statement Data Structures*.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_UNSUPPORTED_PARAM_TYPE`

  The conversion is not supported. Possibly the `buffer_type` value is invalid or is not one of the supported types.

- `CR_OUT_OF_MEMORY`

  Out of memory.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

Example

See the Example in Section 11.10, “mysql_stmt_execute()”.

11.5 mysql_stmt_bind_result()

```
bool
mysql_stmt_bind_result(MYSQL_STMT *stmt,
                       MYSQL_BIND *bind)
```

Description

`mysql_stmt_bind_result()` is used to associate (that is, bind) output columns in the result set to data buffers and length buffers. When `mysql_stmt_fetch()` is called to fetch data, the MySQL client/server protocol places the data for the bound columns into the specified buffers.

All columns must be bound to buffers prior to calling `mysql_stmt_fetch()`. `bind` is the address of an array of `MYSQL_BIND` structures. The client library expects the array to contain one element for each column of the result set. If you do not bind columns to `MYSQL_BIND` structures, `mysql_stmt_fetch()` simply ignores the data fetch. The buffers should be large enough to hold the data values, because the protocol does not return data values in chunks.

A column can be bound or rebound at any time, even after a result set has been partially retrieved. The new binding takes effect the next time `mysql_stmt_fetch()` is called. Suppose that an application binds the columns in a result set and calls `mysql_stmt_fetch()`. The client/server protocol returns data in the bound buffers. Then suppose that the application binds the columns to a different set of buffers. The protocol places data into the newly bound buffers when the next call to `mysql_stmt_fetch()` occurs.

To bind a column, an application calls `mysql_stmt_bind_result()` and passes the type, address, and length of the output buffer into which the value should be stored. Chapter 9, *C API Prepared Statement Data Structures*, describes the members of each `MYSQL_BIND` element and how they should be set to receive output values.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_UNSUPPORTED_PARAM_TYPE`

  The conversion is not supported. Possibly the `buffer_type` value is invalid or is not one of the supported types.

- `CR_OUT_OF_MEMORY`

  Out of memory.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

Example

See the Example in Section 11.11, “mysql_stmt_fetch()”.

11.6 mysql_stmt_close()

```
bool
mysql_stmt_close(MYSQL_STMT *stmt)
```

Description

Closes the prepared statement. `mysql_stmt_close()` also deallocates the statement handler pointed to by `stmt`, which at that point becomes invalid and should no longer be used. For a failed `mysql_stmt_close()` call, do not call `mysql_stmt_error()`, or `mysql_stmt_errno()`, or `mysql_stmt_sqlstate()` to obtain error information because `mysql_stmt_close()` makes the statement handler invalid. Call `mysql_error()`, `mysql_errno()`, or `mysql_sqlstate()` instead.

If the current statement has pending or unread results, this function cancels them so that the next query can be executed.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

Example

See the Example in Section 11.10, “mysql_stmt_execute()”.

11.7 mysql_stmt_data_seek()

```
void
mysql_stmt_data_seek(MYSQL_STMT *stmt,
                     uint64_t offset)
```

Description

Seeks to an arbitrary row in a statement result set. The `offset` value is a row number and should be in the range from `0` to `mysql_stmt_num_rows(stmt)-1`.

This function requires that the statement result set structure contains the entire result of the last executed query, so `mysql_stmt_data_seek()` may be used only in conjunction with `mysql_stmt_store_result()`.

Return Values

None.

Errors

None.

11.8 mysql_stmt_errno()

```
unsigned int
mysql_stmt_errno(MYSQL_STMT *stmt)
```

Description

For the statement specified by `stmt`, `mysql_stmt_errno()` returns the error code for the most recently invoked statement API function that can succeed or fail. A return value of zero means that no error occurred. Client error message numbers are listed in the MySQL `errmsg.h` header file. Server error message numbers are listed in `mysqld_error.h`. Errors also are listed at Error Messages and Common Problems.

If the failed statement API function was `mysql_stmt_close()`, do not call or `mysql_stmt_errno()` to obtain error information because `mysql_stmt_close()` makes the statement handler invalid. Call `mysql_errno()` instead.

Return Values

An error code value. Zero if no error occurred.

Errors

None.

11.9 mysql_stmt_error()

```
const char *
mysql_stmt_error(MYSQL_STMT *stmt)
```

Description

For the statement specified by `stmt`, `mysql_stmt_error()` returns a null-terminated string containing the error message for the most recently invoked statement API function that can succeed or fail. An empty string (`""`) is returned if no error occurred. Either of these two tests can be used to check for an error:

```
if(*mysql_stmt_errno(stmt))
{
  // an error occurred
}

if (mysql_stmt_error(stmt)[0])
{
  // an error occurred
}
```

If the failed statement API function was `mysql_stmt_close()`, do not call `mysql_stmt_error()` to obtain error information because `mysql_stmt_close()` makes the statement handler invalid. Call `mysql_error()` instead.

The language of the client error messages may be changed by recompiling the MySQL client library. You can choose error messages in several different languages.

Return Values

A character string that describes the error. An empty string if no error occurred.

Errors

None.

11.10 mysql_stmt_execute()

```
int
mysql_stmt_execute(MYSQL_STMT *stmt)
```

Description

`mysql_stmt_execute()` executes the prepared query associated with the statement handler. The currently bound parameter marker values are sent to server during this call, and the server replaces the markers with this newly supplied data.

Statement processing following `mysql_stmt_execute()` depends on the type of statement:

- For an `UPDATE`, `DELETE`, or `INSERT`, the number of changed, deleted, or inserted rows can be found by calling `mysql_stmt_affected_rows()`.
- For a statement such as `SELECT` that generates a result set, you must call `mysql_stmt_fetch()` to fetch the data prior to calling any other functions that result in query processing. For more information on how to fetch the results, refer to Section 11.11, “mysql_stmt_fetch()”.

  Do not follow invocation of `mysql_stmt_execute()` with a call to `mysql_store_result()` or `mysql_use_result()`. Those functions are not intended for processing results from prepared statements.

For statements that generate a result set, you can request that `mysql_stmt_execute()` open a cursor for the statement by calling `mysql_stmt_attr_set()` before executing the statement. If you execute a statement multiple times, `mysql_stmt_execute()` closes any open cursor before opening a new one.

Metadata changes to tables or views referred to by prepared statements are detected and cause automatic repreparation of the statement when it is next executed. For more information, see Caching of Prepared Statements and Stored Programs.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_OUT_OF_MEMORY`

  Out of memory.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

Example

The following example demonstrates how to create and populate a table using `mysql_stmt_init()`, `mysql_stmt_prepare()`, `mysql_stmt_param_count()`, `mysql_stmt_bind_param()`, `mysql_stmt_execute()`, and `mysql_stmt_affected_rows()`. The `mysql` variable is assumed to be a valid connection handler. For an example that shows how to retrieve data, see Section 11.11, “mysql_stmt_fetch()”.

```
#define STRING_SIZE 50

#define DROP_SAMPLE_TABLE "DROP TABLE IF EXISTS test_table"
#define CREATE_SAMPLE_TABLE "CREATE TABLE test_table(col1 INT,\
                                                     col2 VARCHAR(40),\
                                                     col3 SMALLINT,\
                                                     col4 TIMESTAMP)"
#define INSERT_SAMPLE "INSERT INTO \
                       test_table(col1,col2,col3) \
                       VALUES(?,?,?)"

MYSQL_STMT    *stmt;
MYSQL_BIND    bind[3];
uint64_t      affected_rows;
int           param_count;
short         small_data;
int           int_data;
char          str_data[STRING_SIZE];
unsigned long str_length;
bool          is_null;

if (mysql_query(mysql, DROP_SAMPLE_TABLE))
{
  fprintf(stderr, " DROP TABLE failed\n");
```

```
  fprintf(stderr, " %s\n", mysql_error(mysql));
  exit(0);
}

if (mysql_query(mysql, CREATE_SAMPLE_TABLE))
{
  fprintf(stderr, " CREATE TABLE failed\n");
  fprintf(stderr, " %s\n", mysql_error(mysql));
  exit(0);
}

/* Prepare an INSERT query with 3 parameters */
/* (the TIMESTAMP column is not named; the server */
/*  sets it to the current date and time) */
stmt = mysql_stmt_init(mysql);
if (!stmt)
{
  fprintf(stderr, " mysql_stmt_init(), out of memory\n");
  exit(0);
}
if (mysql_stmt_prepare(stmt, INSERT_SAMPLE, strlen(INSERT_SAMPLE)))
{
  fprintf(stderr, " mysql_stmt_prepare(), INSERT failed\n");
  fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
  exit(0);
}
fprintf(stdout, " prepare, INSERT successful\n");

/* Get the parameter count from the statement */
param_count= mysql_stmt_param_count(stmt);
fprintf(stdout, " total parameters in INSERT: %d\n", param_count);

if (param_count != 3) /* validate parameter count */
{
  fprintf(stderr, " invalid parameter count returned by MySQL\n");
  exit(0);
}

/* Bind the data for all 3 parameters */

memset(bind, 0, sizeof(bind));

/* INTEGER PARAM */
/* This is a number type, so there is no need
   to specify buffer_length */
bind[0].buffer_type= MYSQL_TYPE_LONG;
bind[0].buffer= (char *)&int_data;
bind[0].is_null= 0;
bind[0].length= 0;

/* STRING PARAM */
bind[1].buffer_type= MYSQL_TYPE_STRING;
bind[1].buffer= (char *)str_data;
bind[1].buffer_length= STRING_SIZE;
bind[1].is_null= 0;
bind[1].length= &str_length;

/* SMALLINT PARAM */
bind[2].buffer_type= MYSQL_TYPE_SHORT;
bind[2].buffer= (char *)&small_data;
bind[2].is_null= &is_null;
bind[2].length= 0;

/* Bind the buffers */
if (mysql_stmt_bind_param(stmt, bind))
{
  fprintf(stderr, " mysql_stmt_bind_param() failed\n");
```

```
  fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
  exit(0);
}

/* Specify the data values for the first row */
int_data= 10;             /* integer */
strncpy(str_data, "MySQL", STRING_SIZE); /* string  */
str_length= strlen(str_data);

/* INSERT SMALLINT data as NULL */
is_null= 1;

/* Execute the INSERT statement - 1*/
if (mysql_stmt_execute(stmt))
{
  fprintf(stderr, " mysql_stmt_execute(), 1 failed\n");
  fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
  exit(0);
}

/* Get the number of affected rows */
affected_rows= mysql_stmt_affected_rows(stmt);
fprintf(stdout, " total affected rows(insert 1): %lu\n",
                (unsigned long) affected_rows);

if (affected_rows != 1) /* validate affected rows */
{
  fprintf(stderr, " invalid affected rows by MySQL\n");
  exit(0);
}

/* Specify data values for second row,
   then re-execute the statement */
int_data= 1000;
strncpy(str_data, "
        The most popular Open Source database",
        STRING_SIZE);
str_length= strlen(str_data);
small_data= 1000;         /* smallint */
is_null= 0;               /* reset */

/* Execute the INSERT statement - 2*/
if (mysql_stmt_execute(stmt))
{
  fprintf(stderr, " mysql_stmt_execute, 2 failed\n");
  fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
  exit(0);
}

/* Get the total rows affected */
affected_rows= mysql_stmt_affected_rows(stmt);
fprintf(stdout, " total affected rows(insert 2): %lu\n",
                (unsigned long) affected_rows);

if (affected_rows != 1) /* validate affected rows */
{
  fprintf(stderr, " invalid affected rows by MySQL\n");
  exit(0);
}

/* Close the statement */
if (mysql_stmt_close(stmt))
{
  /* mysql_stmt_close() invalidates stmt, so call          */
  /* mysql_error(mysql) rather than mysql_stmt_error(stmt) */
  fprintf(stderr, " failed while closing the statement\n");
  fprintf(stderr, " %s\n", mysql_error(mysql));
```

```
  exit(0);
}
```

> **Note**
>
> For complete examples on the use of prepared statement functions, refer to the file `tests/mysql_client_test.c`. This file can be obtained from a MySQL source distribution or from the source repository (see Installing MySQL from Source).

11.11 mysql_stmt_fetch()

```
int
mysql_stmt_fetch(MYSQL_STMT *stmt)
```

Description

`mysql_stmt_fetch()` returns the next row in the result set. It can be called only while the result set exists; that is, after a call to `mysql_stmt_execute()` for a statement such as `SELECT` that produces a result set.

`mysql_stmt_fetch()` returns row data using the buffers bound by `mysql_stmt_bind_result()`. It returns the data in those buffers for all the columns in the current row set and the lengths are returned to the `length` pointer. All columns must be bound by the application before it calls `mysql_stmt_fetch()`.

`mysql_stmt_fetch()` typically occurs within a loop, to ensure that all result set rows are fetched. For example:

```
int status;

while (1)
{
  status = mysql_stmt_fetch(stmt);

  if (status == 1 || status == MYSQL_NO_DATA)
    break;

  /* handle current row here */
}

/* if desired, handle status == 1 case and display error here */
```

By default, result sets are fetched unbuffered a row at a time from the server. To buffer the entire result set on the client, call `mysql_stmt_store_result()` after binding the data buffers and before calling `mysql_stmt_fetch()`.

If a fetched data value is a `NULL` value, the `*is_null` value of the corresponding `MYSQL_BIND` structure contains TRUE (1). Otherwise, the data and its length are returned in the `*buffer` and `*length` elements based on the buffer type specified by the application. Each numeric and temporal type has a fixed length, as listed in the following table. The length of the string types depends on the length of the actual data value, as indicated by `data_length`.

Type	Length
`MYSQL_TYPE_TINY`	1
`MYSQL_TYPE_SHORT`	2
`MYSQL_TYPE_LONG`	4
`MYSQL_TYPE_LONGLONG`	8

Type	Length
MYSQL_TYPE_FLOAT	4
MYSQL_TYPE_DOUBLE	8
MYSQL_TYPE_TIME	sizeof(MYSQL_TIME)
MYSQL_TYPE_DATE	sizeof(MYSQL_TIME)
MYSQL_TYPE_DATETIME	sizeof(MYSQL_TIME)
MYSQL_TYPE_STRING	data length
MYSQL_TYPE_BLOB	data_length

In some cases, you might want to determine the length of a column value before fetching it with mysql_stmt_fetch(). For example, the value might be a long string or BLOB value for which you want to know how much space must be allocated. To accomplish this, use one of these strategies:

- Before invoking mysql_stmt_fetch() to retrieve individual rows, pass STMT_ATTR_UPDATE_MAX_LENGTH to mysql_stmt_attr_set(), then invoke mysql_stmt_store_result() to buffer the entire result on the client side. Setting the STMT_ATTR_UPDATE_MAX_LENGTH attribute causes the maximal length of column values to be indicated by the max_length member of the result set metadata returned by mysql_stmt_result_metadata().
- Invoke mysql_stmt_fetch() with a zero-length buffer for the column in question and a pointer in which the real length can be stored. Then use the real length with mysql_stmt_fetch_column().

```
real_length= 0;

bind[0].buffer= 0;
bind[0].buffer_length= 0;
bind[0].length= &real_length
mysql_stmt_bind_result(stmt, bind);

mysql_stmt_fetch(stmt);
if (real_length > 0)
{
  data= malloc(real_length);
  bind[0].buffer= data;
  bind[0].buffer_length= real_length;
  mysql_stmt_fetch_column(stmt, bind, 0, 0);
}
```

Return Values

Return Value	Description
0	Success, the data has been fetched to application data buffers.
1	Error occurred. Error code and message can be obtained by calling mysql_stmt_errno() and mysql_stmt_error().
MYSQL_NO_DATA	Success, no more data exists
MYSQL_DATA_TRUNCATED	Data truncation occurred

MYSQL_DATA_TRUNCATED is returned when truncation reporting is enabled. To determine which column values were truncated when this value is returned, check the error members of the MYSQL_BIND structures used for fetching values. Truncation reporting is enabled by default, but can be controlled by calling mysql_options() with the MYSQL_REPORT_DATA_TRUNCATION option.

Errors

- CR_COMMANDS_OUT_OF_SYNC

  Commands were executed in an improper order.

  Although mysql_stmt_fetch() can produce this error, it is more likely to occur for the *following* C API call if mysql_stmt_fetch() is not called enough times to read the entire result set (that is, enough times to return MYSQL_NO_DATA).

- CR_OUT_OF_MEMORY

  Out of memory.

- CR_SERVER_GONE_ERROR

  The MySQL server has gone away.

- CR_SERVER_LOST

  The connection to the server was lost during the query.

- CR_UNKNOWN_ERROR

  An unknown error occurred.

- CR_UNSUPPORTED_PARAM_TYPE

  The buffer type is MYSQL_TYPE_DATE, MYSQL_TYPE_TIME, MYSQL_TYPE_DATETIME, or MYSQL_TYPE_TIMESTAMP, but the data type is not DATE, TIME, DATETIME, or TIMESTAMP.

- All other unsupported conversion errors are returned from mysql_stmt_bind_result().

Example

The following example demonstrates how to fetch data from a table using mysql_stmt_result_metadata(), mysql_stmt_bind_result(), and mysql_stmt_fetch(). (This example expects to retrieve the two rows inserted by the example shown in Section 11.10, “mysql_stmt_execute()”.) The mysql variable is assumed to be a valid connection handler.

```
#define STRING_SIZE 50

#define SELECT_SAMPLE "SELECT col1, col2, col3, col4 \
                       FROM test_table"

MYSQL_STMT    *stmt;
MYSQL_BIND    bind[4];
MYSQL_RES     *prepare_meta_result;
MYSQL_TIME    ts;
unsigned long length[4];
int           param_count, column_count, row_count;
short         small_data;
int           int_data;
char          str_data[STRING_SIZE];
bool          is_null[4];
bool          error[4];

/* Prepare a SELECT query to fetch data from test_table */
stmt = mysql_stmt_init(mysql);
if (!stmt)
{
  fprintf(stderr, " mysql_stmt_init(), out of memory\n");
```

```
  exit(0);
}
if (mysql_stmt_prepare(stmt, SELECT_SAMPLE, strlen(SELECT_SAMPLE)))
{
  fprintf(stderr, " mysql_stmt_prepare(), SELECT failed\n");
  fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
  exit(0);
}
fprintf(stdout, " prepare, SELECT successful\n");

/* Get the parameter count from the statement */
param_count= mysql_stmt_param_count(stmt);
fprintf(stdout, " total parameters in SELECT: %d\n", param_count);

if (param_count != 0) /* validate parameter count */
{
  fprintf(stderr, " invalid parameter count returned by MySQL\n");
  exit(0);
}

/* Execute the SELECT query */
if (mysql_stmt_execute(stmt))
{
  fprintf(stderr, " mysql_stmt_execute(), failed\n");
  fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
  exit(0);
}

/* Fetch result set meta information */
prepare_meta_result = mysql_stmt_result_metadata(stmt);
if (!prepare_meta_result)
{
  fprintf(stderr,
         " mysql_stmt_result_metadata(), \
           returned no meta information\n");
  fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
  exit(0);
}

/* Get total columns in the query */
column_count= mysql_num_fields(prepare_meta_result);
fprintf(stdout,
        " total columns in SELECT statement: %d\n",
        column_count);

if (column_count != 4) /* validate column count */
{
  fprintf(stderr, " invalid column count returned by MySQL\n");
  exit(0);
}

/* Bind the result buffers for all 4 columns before fetching them */

memset(bind, 0, sizeof(bind));

/* INTEGER COLUMN */
bind[0].buffer_type= MYSQL_TYPE_LONG;
bind[0].buffer= (char *)&int_data;
bind[0].is_null= &is_null[0];
bind[0].length= &length[0];
bind[0].error= &error[0];

/* STRING COLUMN */
bind[1].buffer_type= MYSQL_TYPE_STRING;
bind[1].buffer= (char *)str_data;
bind[1].buffer_length= STRING_SIZE;
bind[1].is_null= &is_null[1];
```

```
bind[1].length= &length[1];
bind[1].error= &error[1];

/* SMALLINT COLUMN */
bind[2].buffer_type= MYSQL_TYPE_SHORT;
bind[2].buffer= (char *)&small_data;
bind[2].is_null= &is_null[2];
bind[2].length= &length[2];
bind[2].error= &error[2];

/* TIMESTAMP COLUMN */
bind[3].buffer_type= MYSQL_TYPE_TIMESTAMP;
bind[3].buffer= (char *)&ts;
bind[3].is_null= &is_null[3];
bind[3].length= &length[3];
bind[3].error= &error[3];

/* Bind the result buffers */
if (mysql_stmt_bind_result(stmt, bind))
{
  fprintf(stderr, " mysql_stmt_bind_result() failed\n");
  fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
  exit(0);
}

/* Now buffer all results to client (optional step) */
if (mysql_stmt_store_result(stmt))
{
  fprintf(stderr, " mysql_stmt_store_result() failed\n");
  fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
  exit(0);
}

/* Fetch all rows */
row_count= 0;
fprintf(stdout, "Fetching results ...\n");
while (!mysql_stmt_fetch(stmt))
{
  row_count++;
  fprintf(stdout, "  row %d\n", row_count);

  /* column 1 */
  fprintf(stdout, "   column1 (integer)  : ");
  if (is_null[0])
    fprintf(stdout, " NULL\n");
  else
    fprintf(stdout, " %d(%ld)\n", int_data, length[0]);

  /* column 2 */
  fprintf(stdout, "   column2 (string)   : ");
  if (is_null[1])
    fprintf(stdout, " NULL\n");
  else
    fprintf(stdout, " %s(%ld)\n", str_data, length[1]);

  /* column 3 */
  fprintf(stdout, "   column3 (smallint) : ");
  if (is_null[2])
    fprintf(stdout, " NULL\n");
  else
    fprintf(stdout, " %d(%ld)\n", small_data, length[2]);

  /* column 4 */
  fprintf(stdout, "   column4 (timestamp): ");
  if (is_null[3])
    fprintf(stdout, " NULL\n");
  else
```

```
    fprintf(stdout, " %04d-%02d-%02d %02d:%02d:%02d (%ld)\n",
                      ts.year, ts.month, ts.day,
                      ts.hour, ts.minute, ts.second,
                      length[3]);
  fprintf(stdout, "\n");
}

/* Validate rows fetched */
fprintf(stdout, " total rows fetched: %d\n", row_count);
if (row_count != 2)
{
  fprintf(stderr, " MySQL failed to return all rows\n");
  exit(0);
}

/* Free the prepared result metadata */
mysql_free_result(prepare_meta_result);


/* Close the statement */
if (mysql_stmt_close(stmt))
{
  /* mysql_stmt_close() invalidates stmt, so call          */
  /* mysql_error(mysql) rather than mysql_stmt_error(stmt) */
  fprintf(stderr, " failed while closing the statement\n");
  fprintf(stderr, " %s\n", mysql_error(mysql));
  exit(0);
}
```

11.12 mysql_stmt_fetch_column()

```
int
mysql_stmt_fetch_column(MYSQL_STMT *stmt,
                        MYSQL_BIND *bind,
                        unsigned int column,
                        unsigned long offset)
```

Description

Fetches one column from the current result set row. `bind` provides the buffer where data should be placed. It should be set up the same way as for `mysql_stmt_bind_result()`. `column` indicates which column to fetch. The first column is numbered 0. `offset` is the offset within the data value at which to begin retrieving data. This can be used for fetching the data value in pieces. The beginning of the value is offset 0.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_INVALID_PARAMETER_NO`

  Invalid column number.

- `CR_NO_DATA`

  The end of the result set has already been reached.

11.13 mysql_stmt_field_count()

```
unsigned int
mysql_stmt_field_count(MYSQL_STMT *stmt)
```

Description

Returns the number of columns for the most recent statement for the statement handler. This value is zero for statements such as `INSERT` or `DELETE` that do not produce result sets.

`mysql_stmt_field_count()` can be called after you have prepared a statement by invoking `mysql_stmt_prepare()`.

Return Values

An unsigned integer representing the number of columns in a result set.

Errors

None.

11.14 mysql_stmt_free_result()

```
bool
mysql_stmt_free_result(MYSQL_STMT *stmt)
```

Description

Releases memory associated with the result set produced by execution of the prepared statement. If there is a cursor open for the statement, `mysql_stmt_free_result()` closes it.

Return Values

Zero for success. Nonzero if an error occurred.

11.15 mysql_stmt_init()

```
MYSQL_STMT *
mysql_stmt_init(MYSQL *mysql)
```

Description

Creates and returns a `MYSQL_STMT` handler. The handler should be freed with `mysql_stmt_close()`, at which point the handler becomes invalid and should no longer be used.

See also Chapter 9, *C API Prepared Statement Data Structures*, for more information.

Return Values

A pointer to a `MYSQL_STMT` structure in case of success. `NULL` if out of memory.

Errors

- `CR_OUT_OF_MEMORY`

  Out of memory.

11.16 mysql_stmt_insert_id()

```
uint64_t
mysql_stmt_insert_id(MYSQL_STMT *stmt)
```

Description

Returns the value generated for an `AUTO_INCREMENT` column by the prepared `INSERT` or `UPDATE` statement. Use this function after you have executed a prepared `INSERT` statement on a table which contains an `AUTO_INCREMENT` field.

See Section 7.39, “mysql_insert_id()”, for more information.

Return Values

Value for `AUTO_INCREMENT` column which was automatically generated or explicitly set during execution of prepared statement, or value generated by `LAST_INSERT_ID(expr)` function. Return value is undefined if statement does not set `AUTO_INCREMENT` value.

Errors

None.

11.17 mysql_stmt_next_result()

```
int
mysql_stmt_next_result(MYSQL_STMT *mysql)
```

Description

This function is used when you use prepared `CALL` statements to execute stored procedures, which can return multiple result sets. Use a loop that calls `mysql_stmt_next_result()` to determine whether there are more results. If a procedure has `OUT` or `INOUT` parameters, their values will be returned as a single-row result set following any other result sets. The values will appear in the order in which they are declared in the procedure parameter list.

For information about the effect of unhandled conditions on procedure parameters, see Condition Handling and OUT or INOUT Parameters.

`mysql_stmt_next_result()` returns a status to indicate whether more results exist. If `mysql_stmt_next_result()` returns an error, there are no more results.

Before each call to `mysql_stmt_next_result()`, you must call `mysql_stmt_free_result()` for the current result if it produced a result set (rather than just a result status).

After calling `mysql_stmt_next_result()` the state of the connection is as if you had called `mysql_stmt_execute()`. This means that you can call `mysql_stmt_bind_result()`, `mysql_stmt_affected_rows()`, and so forth.

It is also possible to test whether there are more results by calling `mysql_more_results()`. However, this function does not change the connection state, so if it returns true, you must still call `mysql_stmt_next_result()` to advance to the next result.

For an example that shows how to use `mysql_stmt_next_result()`, see Chapter 25, *C API Prepared CALL Statement Support*.

Return Values

Return Value	Description
0	Successful and there are more results
-1	Successful and there are no more results
>0	An error occurred

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

11.18 mysql_stmt_num_rows()

```
uint64_t
mysql_stmt_num_rows(MYSQL_STMT *stmt)
```

Description

Returns the number of rows in the result set.

The use of `mysql_stmt_num_rows()` depends on whether you used `mysql_stmt_store_result()` to buffer the entire result set in the statement handler. If you use `mysql_stmt_store_result()`, `mysql_stmt_num_rows()` may be called immediately. Otherwise, the row count is unavailable unless you count the rows as you fetch them.

`mysql_stmt_num_rows()` is intended for use with statements that return a result set, such as `SELECT`. For statements such as `INSERT`, `UPDATE`, or `DELETE`, the number of affected rows can be obtained with `mysql_stmt_affected_rows()`.

Return Values

The number of rows in the result set.

Errors

None.

11.19 mysql_stmt_param_count()

```
unsigned long
```

```
mysql_stmt_param_count(MYSQL_STMT *stmt)
```

Description

Returns the number of parameter markers present in the prepared statement.

Return Values

An unsigned long integer representing the number of parameters in a statement.

Errors

None.

Example

See the Example in Section 11.10, “mysql_stmt_execute()”.

11.20 mysql_stmt_param_metadata()

```
MYSQL_RES *
mysql_stmt_param_metadata(MYSQL_STMT *stmt)
```

This function currently does nothing.

11.21 mysql_stmt_prepare()

```
int
mysql_stmt_prepare(MYSQL_STMT *stmt,
                   const char *stmt_str,
                   unsigned long length)
```

Description

Given the statement handler returned by `mysql_stmt_init()`, prepares the SQL statement pointed to by the string `stmt_str` and returns a status value. The string length should be given by the `length` argument. The string must consist of a single SQL statement. You should not add a terminating semicolon (`;`) or `\g` to the statement.

The application can include one or more parameter markers in the SQL statement by embedding question mark (`?`) characters into the SQL string at the appropriate positions.

The markers are legal only in certain places in SQL statements. For example, they are permitted in the `VALUES()` list of an `INSERT` statement (to specify column values for a row), or in a comparison with a column in a `WHERE` clause to specify a comparison value. However, they are not permitted for identifiers (such as table or column names), or to specify both operands of a binary operator such as the `=` equal sign. The latter restriction is necessary because it would be impossible to determine the parameter type. In general, parameters are legal only in Data Manipulation Language (DML) statements, and not in Data Definition Language (DDL) statements.

The parameter markers must be bound to application variables using `mysql_stmt_bind_param()` before executing the statement.

Metadata changes to tables or views referred to by prepared statements are detected and cause automatic repreparation of the statement when it is next executed. For more information, see Caching of Prepared Statements and Stored Programs.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_OUT_OF_MEMORY`

  Out of memory.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

If the prepare operation was unsuccessful (that is, `mysql_stmt_prepare()` returns nonzero), the error message can be obtained by calling `mysql_stmt_error()`.

Example

See the Example in Section 11.10, “mysql_stmt_execute()”.

11.22 mysql_stmt_reset()

```
bool
mysql_stmt_reset(MYSQL_STMT *stmt)
```

Description

Resets a prepared statement on client and server to state after prepare. It resets the statement on the server, data sent using `mysql_stmt_send_long_data()`, unbuffered result sets and current errors. It does not clear bindings or stored result sets. Stored result sets will be cleared when executing the prepared statement (or closing it).

To re-prepare the statement with another query, use `mysql_stmt_prepare()`.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- CR_SERVER_GONE_ERROR

  The MySQL server has gone away.

- CR_SERVER_LOST

  The connection to the server was lost during the query

- CR_UNKNOWN_ERROR

  An unknown error occurred.

11.23 mysql_stmt_result_metadata()

```
MYSQL_RES *
mysql_stmt_result_metadata(MYSQL_STMT *stmt)
```

Description

mysql_stmt_result_metadata() is used to obtain result set metadata for a prepared statement. Its use requires that these conditions be satisified:

- The statement when executed by mysql_stmt_execute() does produce a result set.
- The client has not suppressed metadata as described in Chapter 27, *C API Optional Result Set Metadata*.
-

mysql_stmt_result_metadata() may be called after preparing the statement with mysql_stmt_prepare() and before closing the statement handler. The result set metadata returned by mysql_stmt_result_metadata() is in the form of a pointer to a MYSQL_RES structure that can be used to process the meta information such as number of fields and individual field information. This result set pointer can be passed as an argument to any of the field-based API functions that process result set metadata, such as:

- mysql_num_fields()
- mysql_fetch_field()
- mysql_fetch_field_direct()
- mysql_fetch_fields()
- mysql_field_count()
- mysql_field_seek()
- mysql_field_tell()
- mysql_free_result()

When you are done with the metadata result set structure, free it by passing it to mysql_free_result(). This is similar to the way you free a result set structure obtained from a call to mysql_store_result().

If the executed statement is a CALL statement, it may produce multiple result sets. In this case, do not call mysql_stmt_result_metadata() immediately after mysql_stmt_prepare(). Instead, check the

metadata for each result set separately after calling `mysql_stmt_execute()`. For an example of this technique, see Chapter 25, *C API Prepared CALL Statement Support*.

The result set returned by `mysql_stmt_result_metadata()` contains only metadata. It does not contain any row results. To obtain the row results, use the statement handler with `mysql_stmt_fetch()` after executing the statement with `mysql_stmt_execute()`, as usual.

Return Values

A `MYSQL_RES` result structure. `NULL` if no meta information exists for the prepared statement.

Errors

- `CR_OUT_OF_MEMORY`

  Out of memory.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

Example

See the Example in Section 11.11, “mysql_stmt_fetch()”.

11.24 mysql_stmt_row_seek()

```
MYSQL_ROW_OFFSET
mysql_stmt_row_seek(MYSQL_STMT *stmt,
                    MYSQL_ROW_OFFSET offset)
```

Description

Sets the row cursor to an arbitrary row in a statement result set. The `offset` value is a row offset that should be a value returned from `mysql_stmt_row_tell()` or from `mysql_stmt_row_seek()`. This value is not a row number; if you want to seek to a row within a result set by number, use `mysql_stmt_data_seek()` instead.

This function requires that the result set structure contains the entire result of the query, so `mysql_stmt_row_seek()` may be used only in conjunction with `mysql_stmt_store_result()`.

Return Values

The previous value of the row cursor. This value may be passed to a subsequent call to `mysql_stmt_row_seek()`.

Errors

None.

11.25 mysql_stmt_row_tell()

```
MYSQL_ROW_OFFSET
mysql_stmt_row_tell(MYSQL_STMT *stmt)
```

Description

Returns the current position of the row cursor for the last `mysql_stmt_fetch()`. This value can be used as an argument to `mysql_stmt_row_seek()`.

You should use `mysql_stmt_row_tell()` only after `mysql_stmt_store_result()`.

Return Values

The current offset of the row cursor.

Errors

None.

11.26 mysql_stmt_send_long_data()

```
bool
mysql_stmt_send_long_data(MYSQL_STMT *stmt,
                          unsigned int parameter_number,
                          const char *data,
                          unsigned long length)
```

Description

Enables an application to send parameter data to the server in pieces (or “chunks”). Call this function after `mysql_stmt_bind_param()` and before `mysql_stmt_execute()`. It can be called multiple times to send the parts of a character or binary data value for a column, which must be one of the `TEXT` or `BLOB` data types.

`parameter_number` indicates which parameter to associate the data with. Parameters are numbered beginning with 0. `data` is a pointer to a buffer containing data to be sent, and `length` indicates the number of bytes in the buffer.

> **Note**
>
> The next `mysql_stmt_execute()` call ignores the bind buffer for all parameters that have been used with `mysql_stmt_send_long_data()` since last `mysql_stmt_execute()` or `mysql_stmt_reset()`.

To reset/forget the sent data, call `mysql_stmt_reset()`. See Section 11.22, “mysql_stmt_reset()”.

The `max_allowed_packet` system variable controls the maximum size of parameter values that can be sent with `mysql_stmt_send_long_data()`.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_INVALID_BUFFER_USE`

  The parameter does not have a string or binary type.

- CR_INVALID_PARAMETER_NO

  Invalid parameter number.

- CR_COMMANDS_OUT_OF_SYNC

  Commands were executed in an improper order.

- CR_SERVER_GONE_ERROR

  The MySQL server has gone away.

- CR_OUT_OF_MEMORY

  Out of memory.

- CR_UNKNOWN_ERROR

  An unknown error occurred.

Example

The following example demonstrates how to send the data for a TEXT column in chunks. It inserts the data value 'MySQL - The most popular Open Source database' into the text_column column. The mysql variable is assumed to be a valid connection handler.

```
#define INSERT_QUERY "INSERT INTO \
                      test_long_data(text_column) VALUES(?)"

MYSQL_BIND bind[1];
long       length;

stmt = mysql_stmt_init(mysql);
if (!stmt)
{
  fprintf(stderr, " mysql_stmt_init(), out of memory\n");
  exit(0);
}
if (mysql_stmt_prepare(stmt, INSERT_QUERY, strlen(INSERT_QUERY)))
{
  fprintf(stderr, "\n mysql_stmt_prepare(), INSERT failed");
  fprintf(stderr, "\n %s", mysql_stmt_error(stmt));
  exit(0);
}
 memset(bind, 0, sizeof(bind));
 bind[0].buffer_type= MYSQL_TYPE_STRING;
 bind[0].length= &length;
 bind[0].is_null= 0;

/* Bind the buffers */
if (mysql_stmt_bind_param(stmt, bind))
{
  fprintf(stderr, "\n param bind failed");
  fprintf(stderr, "\n %s", mysql_stmt_error(stmt));
  exit(0);
}

 /* Supply data in chunks to server */
 if (mysql_stmt_send_long_data(stmt,0,"MySQL",5))
{
  fprintf(stderr, "\n send_long_data failed");
  fprintf(stderr, "\n %s", mysql_stmt_error(stmt));
  exit(0);
```

```
}

 /* Supply the next piece of data */
 if (mysql_stmt_send_long_data(stmt,0,
           " - The most popular Open Source database",40))
{
  fprintf(stderr, "\n send_long_data failed");
  fprintf(stderr, "\n %s", mysql_stmt_error(stmt));
  exit(0);
}

 /* Now, execute the query */
 if (mysql_stmt_execute(stmt))
{
  fprintf(stderr, "\n mysql_stmt_execute failed");
  fprintf(stderr, "\n %s", mysql_stmt_error(stmt));
  exit(0);
}
```

11.27 mysql_stmt_sqlstate()

```
const char *
mysql_stmt_sqlstate(MYSQL_STMT *stmt)
```

Description

For the statement specified by `stmt`, `mysql_stmt_sqlstate()` returns a null-terminated string containing the SQLSTATE error code for the most recently invoked prepared statement API function that can succeed or fail. The error code consists of five characters. `"00000"` means “no error.” The values are specified by ANSI SQL and ODBC. For a list of possible values, see Error Messages and Common Problems.

Not all MySQL errors are mapped to SQLSTATE codes. The value `"HY000"` (general error) is used for unmapped errors.

If the failed statement API function was `mysql_stmt_close()`, do not call `mysql_stmt_sqlstate()` to obtain error information because `mysql_stmt_close()` makes the statement handler invalid. Call `mysql_sqlstate()` instead.

Return Values

A null-terminated character string containing the SQLSTATE error code.

11.28 mysql_stmt_store_result()

```
int
mysql_stmt_store_result(MYSQL_STMT *stmt)
```

Description

Result sets are produced by calling `mysql_stmt_execute()` to executed prepared statements for SQL statements such as `SELECT`, `SHOW`, `DESCRIBE`, and `EXPLAIN`. By default, result sets for successfully executed prepared statements are not buffered on the client and `mysql_stmt_fetch()` fetches them one at a time from the server. To cause the complete result set to be buffered on the client, call `mysql_stmt_store_result()` after binding data buffers with `mysql_stmt_bind_result()` and before calling `mysql_stmt_fetch()` to fetch rows. (For an example, see Section 11.11, “mysql_stmt_fetch()”.)

`mysql_stmt_store_result()` is optional for result set processing, unless you will call `mysql_stmt_data_seek()`, `mysql_stmt_row_seek()`, or `mysql_stmt_row_tell()`. Those functions require a seekable result set.

It is unnecessary to call `mysql_stmt_store_result()` after executing an SQL statement that does not produce a result set, but if you do, it does not harm or cause any notable performance problem. You can detect whether the statement produced a result set by checking if `mysql_stmt_result_metadata()` returns `NULL`. For more information, refer to Section 11.23, “mysql_stmt_result_metadata()”.

> **Note**
>
> MySQL does not by default calculate `MYSQL_FIELD->max_length` for all columns in `mysql_stmt_store_result()` because calculating this would slow down `mysql_stmt_store_result()` considerably and most applications do not need `max_length`. If you want `max_length` to be updated, you can call `mysql_stmt_attr_set(MYSQL_STMT, STMT_ATTR_UPDATE_MAX_LENGTH, &flag)` to enable this. See Section 11.3, “mysql_stmt_attr_set()”.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_COMMANDS_OUT_OF_SYNC`

  Commands were executed in an improper order.

- `CR_OUT_OF_MEMORY`

  Out of memory.

- `CR_SERVER_GONE_ERROR`

  The MySQL server has gone away.

- `CR_SERVER_LOST`

  The connection to the server was lost during the query.

- `CR_UNKNOWN_ERROR`

  An unknown error occurred.

Chapter 12 C API Asynchronous Interface

As of MySQL 8.0.16, the C API includes asynchronous functions that enable nonblocking communication with the MySQL server. Asynchronous functions enable development of applications that differ from the query processing model based on synchronous functions that block if reads from or writes to the server connection must wait. Using the asynchronous functions, an application can check whether work on the server connection is ready to proceed. If not, the application can perform other work before checking again later.

For example, an application might open multiple connections to the server and use them to submit multiple statements for execution. The application then can poll the connections to see which of them have results to be fetched, while doing other work.

> **Note**
>
> As just indicated, execution of multiple simultaneous statements should be done using multiple connections and executing one statement per connection. The asynchronous interface is not intended for executing multiple simultaneous statements per connection. What it enables is that applications can do other work rather than waiting for server operations to complete.

This section describes the C API asynchronous interface. In this discussion, asynchronous and nonblocking are used as synonyms, as are synchronous and blocking.

The asynchronous C API functions cover operations that might otherwise block when reading to or writing from the server connection: The initial connection operation, sending a query, reading the result, and so forth. Each asynchronous function has the same name as its synchronous counterpart, plus a `_nonblocking` suffix:

- `mysql_fetch_row_nonblocking()`: Asynchronously fetches the next row from the result set.
- `mysql_free_result_nonblocking()`: Asynchronously frees memory used by a result set.
- `mysql_next_result_nonblocking()`: Asynchronously returns/initiates the next result in multiple-result executions.
- `mysql_real_connect_nonblocking()`: Asynchronously connects to a MySQL server.
- `mysql_real_query_nonblocking()`: Asynchronously executes an SQL query specified as a counted string.
- `mysql_store_result_nonblocking()`: Asynchronously retrieves a complete result set to the client.

Applications can mix asynchronous and synchronous functions if there are operations that need not be done asynchronously or for which the asynchronous functions do not apply.

The following discussion describes in more detail how to use asynchronous C API functions.

- Asynchronous Function Calling Conventions
- Example Program
- Asynchronous Function Restrictions

Asynchronous Function Calling Conventions

All asynchronous C API functions return an `enum net_async_status` value. The return value can be one of the following values to indicate operation status:

- NET_ASYNC_NOT_READY: The operation is still in progress and not yet complete.
- NET_ASYNC_COMPLETE: The operation completed successfully.
- NET_ASYNC_ERROR: The operation terminated in error.
- NET_ASYNC_COMPLETE_NO_MORE_RESULTS: The operation completed successfully and no more results are available. This status applies only to mysql_next_result_nonblocking().

In general, to use an asynchronous function, do this:

- Call the function repeatedly until it no longer returns a status of NET_ASYNC_NOT_READY.
- Check whether the final status indicates successful completion (NET_ASYNC_COMPLETE) or an error (NET_ASYNC_ERROR).

The following examples illustrate some typical calling patterns. *function(args)* represents an asynchronous function and its argument list.

- If it is desirable to perform other processing while the operation is in progress:

```
enum net_async_status status;

status = function(args);
while (status == NET_ASYNC_NOT_READY) {
  /* perform other processing */
  other_processing ();
  /* invoke same function and arguments again */
  status = function(args);
}
if (status == NET_ASYNC_ERROR) {
  /* call failed; handle error */
} else {
  /* call successful; handle result */
}
```

- If there is no need to perform other processing while the operation is in progress:

```
enum net_async_status status;

while ((status = function(args)) == NET_ASYNC_NOT_READY)
  ; /* empty loop */
if (status == NET_ASYNC_ERROR) {
  /* call failed; handle error */
} else {
  /* call successful; handle result */
}
```

- If the function success/failure result does not matter and you want to ensure only that the operation has completed:

```
while (function (args) != NET_ASYNC_COMPLETE)
  ; /* empty loop */
```

For mysql_next_result_nonblocking(), it is also necessary to account for the NET_ASYNC_COMPLETE_NO_MORE_RESULTS status, which indicates that the operation completed successfully and no more results are available. Use it like this:

```
while ((status = mysql_next_result_nonblocking()) != NET_ASYNC_COMPLETE) {
  if (status == NET_ASYNC_COMPLETE_NO_MORE_RESULTS) {
    /* no more results */
  }
  else if (status == NET_ASYNC_ERROR) {
    /* handle error by calling mysql_error(); */
```

```
    break;
  }
}
```

In most cases, arguments for the asynchronous functions are the same as for the corresponding synchronous functions. Exceptions are `mysql_fetch_row_nonblocking()` and `mysql_store_result_nonblocking()`, each of which takes an extra argument compared to its synchronous counterpart. For details, see Section 15.1, “mysql_fetch_row_nonblocking()”, and Section 15.6, “mysql_store_result_nonblocking()”.

Example Program

This section shows an example C++ program that illustrates use of asynchronous C API functions.

To set up the SQL objects used by the program, execute the following statements. Substitute a different database or user as desired; in this case, you will need to make some adjustments to the program as well.

```
CREATE DATABASE db;
USE db;
CREATE TABLE test_table (id INT NOT NULL);
INSERT INTO test_table VALUES (10), (20), (30);

CREATE USER 'testuser'@'localhost' IDENTIFIED BY 'testpass';
GRANT ALL ON db.* TO 'testuser'@'localhost';
```

Create a file named `async_app.cc` containing the following program. Adjust the connection parameters as necessary.

```
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <mysql.h>
#include <mysqld_error.h>

using namespace std;

/* change following connection parameters as necessary */
static const char * c_host = "localhost";
static const char * c_user = "testuser";
static const char * c_auth = "testpass";
static int          c_port = 3306;
static const char * c_sock = "/usr/local/mysql/mysql.sock";
static const char * c_dbnm = "db";

void perform_arithmetic() {
  cout<<"dummy function invoked\n";
  for (int i = 0; i < 1000; i++)
    i*i;
}

int main(int argc, char ** argv)
{
  MYSQL *mysql_local;
  MYSQL_RES *result;
  MYSQL_ROW row;
  net_async_status status;
  const char *stmt_text;

  if (!(mysql_local = mysql_init(NULL))) {
    cout<<"mysql_init() failed\n";
    exit(1);
  }
  while ((status = mysql_real_connect_nonblocking(mysql_local, c_host, c_user,
                                                  c_auth, c_dbnm, c_port,
                                                  c_sock, 0))
```

```
          == NET_ASYNC_NOT_READY)
  ; /* empty loop */
if (status == NET_ASYNC_ERROR) {
  cout<<"mysql_real_connect_nonblocking() failed\n";
  exit(1);
}

/* run query asynchronously */
stmt_text = "SELECT * FROM test_table ORDER BY id";
status = mysql_real_query_nonblocking(mysql_local, stmt_text,
                                      (unsigned long)strlen(stmt_text));
/* do some other task before checking function result */
perform_arithmetic();
while (status == NET_ASYNC_NOT_READY) {
  status = mysql_real_query_nonblocking(mysql_local, stmt_text,
                                        (unsigned long)strlen(stmt_text));
  perform_arithmetic();
}
if (status == NET_ASYNC_ERROR) {
  cout<<"mysql_real_query_nonblocking() failed\n";
  exit(1);
}

/* retrieve query result asynchronously */
status = mysql_store_result_nonblocking(mysql_local, &result);
/* do some other task before checking function result */
perform_arithmetic();
while (status == NET_ASYNC_NOT_READY) {
  status = mysql_store_result_nonblocking(mysql_local, &result);
  perform_arithmetic();
}
if (status == NET_ASYNC_ERROR) {
  cout<<"mysql_store_result_nonblocking() failed\n";
  exit(1);
}
if (result == NULL) {
  cout<<"mysql_store_result_nonblocking() found 0 records\n";
  exit(1);
}

/* fetch a row synchronously */
row = mysql_fetch_row(result);
if (row != NULL && strcmp(row[0], "10") == 0)
  cout<<"ROW: " << row[0] << "\n";
else
  cout<<"incorrect result fetched\n";

/* fetch a row asynchronously, but without doing other work */
while (mysql_fetch_row_nonblocking(result, &row) != NET_ASYNC_COMPLETE)
  ; /* empty loop */
/* 2nd row fetched */
if (row != NULL && strcmp(row[0], "20") == 0)
  cout<<"ROW: " << row[0] << "\n";
else
  cout<<"incorrect result fetched\n";

/* fetch a row asynchronously, doing other work while waiting */
status = mysql_fetch_row_nonblocking(result, &row);
/* do some other task before checking function result */
perform_arithmetic();
while (status != NET_ASYNC_COMPLETE) {
  status = mysql_fetch_row_nonblocking(result, &row);
  perform_arithmetic();
}
/* 3rd row fetched */
if (row != NULL && strcmp(row[0], "30") == 0)
  cout<<"ROW: " << row[0] << "\n";
```

```
  else
    cout<<"incorrect result fetched\n";

  /* fetch a row asynchronously (no more rows expected) */
  while ((status = mysql_fetch_row_nonblocking(result, &row))
           != NET_ASYNC_COMPLETE)
    ; /* empty loop */
  if (row == NULL)
    cout <<"No more rows to process.\n";
  else
    cout <<"More rows found than expected.\n";

  /* free result set memory asynchronously */
  while (mysql_free_result_nonblocking(result) != NET_ASYNC_COMPLETE)
    ; /* empty loop */

  mysql_close(mysql_local);
}
```

Compile the program using a command similar to this; adjust the compiler and options as necessary:

```
gcc -g async_app.cc -std=c++11 \
  -I/usr/local/mysql/include \
  -o async_app -L/usr/lib64/ -lstdc++ \
  -L/usr/local/mysql/lib/ -lmysqlclient
```

Run the program. The results should be similar to what you see here, although you might see a varying number of `dummy function invoked` instances.

```
dummy function invoked
dummy function invoked
ROW: 10
ROW: 20
dummy function invoked
ROW: 30
No more rows to process.
```

To experiment with the program, add and remove rows from `test_table`, running the program again after each change.

Asynchronous Function Restrictions

These restrictions apply to the use of asynchronous C API functions:

- `mysql_real_connect_nonblocking()` can be used only for accounts that authenticate with one of these authentication plugins: `mysql_native_password`, `sha256_password`, or `caching_sha2_password`.
- `mysql_real_connect_nonblocking()` can be used only to establish TCP/IP or Unix socket file connections.
- These statements are not supported and must be processed using synchronous C API functions: `LOAD DATA`, `LOAD XML`.
- Input arguments passed to an asynchronous C API call that initiates a nonblocking operation may remain in use until the operation terminates later, and should not be reused until termination occurs.
- Protocol compression is not supported for asynchronous C API functions.

Chapter 13 C API Asynchronous Interface Data Structures

This section describes data structures specific to asynchronous C API functions. For information about general-purpose C API data structures, see Chapter 5, *C API Data Structures*.

- `enum net_async_status`

  The enumeration type used to express the return status of asynchronous C API functions. The following table shows the permitted status values.

Enumeration Status Value	Description
`NET_ASYNC_COMPLETE`	Asynchronous operation is complete
`NET_ASYNC_NOT_READY`	Asynchronous operation is still in progress
`NET_ASYNC_ERROR`	Asynchronous operation terminated in error
`NET_ASYNC_COMPLETE_NO_MORE_RESULTS`	For `mysql_next_result_nonblocking()`; indicates no more results available

  For more information, see Chapter 12, *C API Asynchronous Interface*.

Chapter 14 C API Asynchronous Function Overview

The following list summarizes the functions available for asynchronous interaction with the MySQL server. For greater detail, see the descriptions in Chapter 15, *C API Asynchronous Function Descriptions*.

- `mysql_fetch_row_nonblocking()`: Asynchronously fetches the next row from the result set.
- `mysql_free_result_nonblocking()`: Asynchronously frees memory used by a result set.
- `mysql_next_result_nonblocking()`: Asynchronously returns/initiates the next result in multiple-result executions.
- `mysql_real_connect_nonblocking()`: Asynchronously connects to a MySQL server.
- `mysql_real_query_nonblocking()`: Asynchronously executes an SQL query specified as a counted string.
- `mysql_store_result_nonblocking()`: Asynchronously retrieves a complete result set to the client.

Chapter 15 C API Asynchronous Function Descriptions

Table of Contents

To interact asynchronously with the MySQL server, use the functions described in the following sections. For descriptions of their synchronous counterparts, see Chapter 7, *C API Function Descriptions*.

15.1 mysql_fetch_row_nonblocking()

```
enum net_async_status
mysql_fetch_row_nonblocking(MYSQL_RES *result,
                            MYSQL_ROW *row)
```

Description

> **Note**
>
> `mysql_fetch_row_nonblocking()` is an asynchronous function. It is the counterpart of the `mysql_fetch_row()` synchronous function, for use by applications that require asynchronous communication with the server. For general information about writing asynchronous C API applications, see Chapter 12, *C API Asynchronous Interface*.

`mysql_fetch_row_nonblocking()` is used similarly to `mysql_fetch_row()`. For details about the latter, see Section 7.22, “mysql_fetch_row()”. The two functions differ as follows:

- `mysql_fetch_row()` returns a `MYSQL_ROW` value containing the next row, or `NULL`. The meaning of a `NULL` return depends on which function was called preceding `mysql_fetch_row()`:
  - When used after `mysql_store_result()` or `mysql_store_result_nonblocking()`, `mysql_fetch_row()` returns `NULL` if there are no more rows to retrieve.
  - When used after `mysql_use_result()`, `mysql_fetch_row()` returns `NULL` if there are no more rows to retrieve or an error occurred.
- `mysql_fetch_row_nonblocking()` returns an `enum net_async_status` status indicator and takes a second `row` argument that provides a pointer to a `MYSQL_ROW` value. When the return status is `NET_ASYNC_COMPLETE`, the `row` argument is a pointer to a `MYSQL_ROW` value containing the next row, or `NULL`. The meaning of `NULL` depends on which function was called preceding `mysql_fetch_row_nonblocking()`:
  - When used after `mysql_store_result()` or `mysql_store_result_nonblocking()`, the `row` argument is `NULL` if there are no more rows to retrieve.
  - When used after `mysql_use_result()`, the `row` argument is `NULL` if there are no more rows to retrieve or an error occurred.

`mysql_fetch_row_nonblocking()` was added in MySQL 8.0.16.

Return Values

Returns an `enum net_async_status` value. See the description in Chapter 13, *C API Asynchronous Interface Data Structures*. A `NET_ASYNC_ERROR` return status indicates an error.

Example

See Chapter 12, *C API Asynchronous Interface*.

15.2 mysql_free_result_nonblocking()

```
enum net_async_status
mysql_free_result_nonblocking(MYSQL_RES *result)
```

Description

> **Note**
>
> `mysql_free_result_nonblocking()` is an asynchronous function. It is the counterpart of the `mysql_free_result()` synchronous function, for use by applications that require asynchronous communication with the server. For general information about writing asynchronous C API applications, see Chapter 12, *C API Asynchronous Interface*.

`mysql_free_result_nonblocking()` is used similarly to `mysql_free_result()`. For details about the latter, see Section 7.26, "mysql_free_result()". The two functions differ as follows:

- `mysql_free_result()` does not return a value.
- `mysql_free_result_nonblocking()` returns an `enum net_async_status` status indicator.

`mysql_free_result_nonblocking()` was added in MySQL 8.0.16.

Return Values

Returns an `enum net_async_status` value. See the description in Chapter 13, *C API Asynchronous Interface Data Structures*. A `NET_ASYNC_ERROR` return status indicates an error.

Example

See Chapter 12, *C API Asynchronous Interface*.

15.3 mysql_next_result_nonblocking()

```
enum net_async_status
mysql_next_result_nonblocking(MYSQL *mysql)
```

Description

> **Note**
>
> `mysql_next_result_nonblocking()` is an asynchronous function. It is the counterpart of the `mysql_next_result()` synchronous function, for use by applications that require asynchronous communication with the server. For general

> information about writing asynchronous C API applications, see Chapter 12, *C API Asynchronous Interface*.

`mysql_next_result_nonblocking()` is used similarly to `mysql_next_result()`. For details about the latter, see Section 7.48, "mysql_next_result()". The two functions differ as follows:

- `mysql_next_result()` returns an integer status indicator.
- `mysql_next_result_nonblocking()` returns an `enum net_async_status` status indicator.

`mysql_next_result_nonblocking()` was added in MySQL 8.0.16.

Return Values

Returns an `enum net_async_status` value. See the description in Chapter 13, *C API Asynchronous Interface Data Structures*. A `NET_ASYNC_COMPLETE_NO_MORE_RESULTS` return status indicates there are no more results available. A `NET_ASYNC_ERROR` return status indicates an error.

Example

See Chapter 12, *C API Asynchronous Interface*.

15.4 mysql_real_connect_nonblocking()

```
enum net_async_status
mysql_real_connect_nonblocking(MYSQL *mysql,
                               const char *host,
                               const char *user,
                               const char *passwd,
                               const char *db,
                               unsigned int port,
                               const char *unix_socket,
                               unsigned long
                               client_flag)
```

Description

> **Note**
>
> `mysql_real_connect_nonblocking()` is an asynchronous function. It is the counterpart of the `mysql_real_connect()` synchronous function, for use by applications that require asynchronous communication with the server. For general information about writing asynchronous C API applications, see Chapter 12, *C API Asynchronous Interface*.

`mysql_real_connect_nonblocking()` is used similarly to `mysql_real_connect()`. For details about the latter, see Section 7.55, "mysql_real_connect()". The two functions differ as follows:

- `mysql_real_connect()` returns a connection handler or `NULL`.
- `mysql_real_connect_nonblocking()` returns an `enum net_async_status` status indicator.

`mysql_real_connect_nonblocking()` was added in MySQL 8.0.16.

Return Values

Returns an `enum net_async_status` value. See the description in Chapter 13, *C API Asynchronous Interface Data Structures*. A `NET_ASYNC_ERROR` return status indicates an error.

Example

See Chapter 12, *C API Asynchronous Interface*.

15.5 mysql_real_query_nonblocking()

```
enum net_async_status
mysql_real_query_nonblocking(MYSQL *mysql,
                             const char *stmt_str,
                             unsigned long length)
```

Description

> **Note**
>
> `mysql_real_query_nonblocking()` is an asynchronous function. It is the counterpart of the `mysql_real_query()` synchronous function, for use by applications that require asynchronous communication with the server. For general information about writing asynchronous C API applications, see Chapter 12, *C API Asynchronous Interface*.

`mysql_real_query_nonblocking()` is used similarly to `mysql_real_query()`. For details about the latter, see Section 7.59, "mysql_real_query()". The two functions differ as follows:

- `mysql_real_query()` returns an integer status indicator.
- `mysql_real_query_nonblocking()` returns an `enum net_async_status` status indicator.

`mysql_real_query_nonblocking()` was added in MySQL 8.0.16.

Return Values

Returns an `enum net_async_status` value. See the description in Chapter 13, *C API Asynchronous Interface Data Structures*. A `NET_ASYNC_ERROR` return status indicates an error.

Example

See Chapter 12, *C API Asynchronous Interface*.

15.6 mysql_store_result_nonblocking()

```
enum net_async_status
mysql_store_result_nonblocking(MYSQL *mysql,
                               MYSQL_RES **result)
```

Description

> **Note**
>
> `mysql_store_result_nonblocking()` is an asynchronous function. It is the counterpart of the `mysql_store_result()` synchronous function, for use by applications that require asynchronous communication with the server. For general information about writing asynchronous C API applications, see Chapter 12, *C API Asynchronous Interface*.

`mysql_store_result_nonblocking()` is used similarly to `mysql_store_result()`. For details about the latter, see Section 7.81, "mysql_store_result()". The two functions differ as follows:

- `mysql_store_result()` returns a pointer to a `MYSQL_RESULT` value that contains the result set, or `NULL` if there is no result set or an error occurred.
- `mysql_store_result_nonblocking()` returns an `enum net_async_status` status indicator and takes a second `result` argument that is the address of a pointer to a `MYSQL_RESULT` into which to store the result set. When the return status is `NET_ASYNC_COMPLETE`, the `result` argument is `NULL` if there is no result set or an error occurred.

`mysql_store_result_nonblocking()` was added in MySQL 8.0.16.

Return Values

Returns an `enum net_async_status` value. See the description in Chapter 13, *C API Asynchronous Interface Data Structures*. A `NET_ASYNC_ERROR` return status indicates an error.

When the return status is `NET_ASYNC_COMPLETE`, the `result` argument is `NULL` if there is no result set or an error occurred. To determine whether an error occurred, check whether `mysql_error()` returns a nonempty string, `mysql_errno()` returns nonzero, or `mysql_field_count()` returns zero.

Example

See Chapter 12, *C API Asynchronous Interface*.

Chapter 16 C API Threaded Function Descriptions

Table of Contents

To create a threaded client, use the functions described in the following sections. See also Section 4.3, “Writing C API Threaded Client Programs”.

16.1 mysql_thread_end()

```
void
mysql_thread_end(void)
```

Description

Call this function as necessary before calling `pthread_exit()` to free memory allocated by `mysql_thread_init()`:

- For release/production builds without debugging support enabled, `mysql_thread_end()` need not be called.
- For debug builds, `mysql_thread_init()` allocates debugging information for the DBUG package (see The DBUG Package). `mysql_thread_end()` must be called for each `mysql_thread_init()` call to avoid a memory leak.

`mysql_thread_end()` *is not invoked automatically by the client library*.

Return Values

None.

16.2 mysql_thread_init()

```
bool
mysql_thread_init(void)
```

Description

This function must be called early within each created thread to initialize thread-specific variables. However, it may be unnecessarily to invoke it explicitly. Calling `mysql_thread_init()` is automatically handled by `mysql_init()`, `mysql_library_init()`, `mysql_server_init()`, and `mysql_connect()`. If you invoke any of those functions, `mysql_thread_init()` is called for you.

Return Values

Zero for success. Nonzero if an error occurred.

16.3 mysql_thread_safe()

```
unsigned
```

```
int mysql_thread_safe(void)
```

Description

This function indicates whether the client library is compiled as thread-safe.

Return Values

1 if the client library is thread-safe, 0 otherwise.

Chapter 17 C API Client Plugin Functions

Table of Contents

This section describes functions used for the client-side plugin API. They enable management of client plugins. For a description of the `st_mysql_client_plugin` structure used by these functions, see Client Plugin Descriptors.

It is unlikely that a client program needs to call the functions in this section. For example, a client that supports the use of authentication plugins normally causes a plugin to be loaded by calling `mysql_options()` to set the `MYSQL_DEFAULT_AUTH` and `MYSQL_PLUGIN_DIR` options:

```
char *plugin_dir = "path_to_plugin_dir";
char *default_auth = "plugin_name";

/* ... process command-line options ... */

mysql_options(&mysql, MYSQL_PLUGIN_DIR, plugin_dir);
mysql_options(&mysql, MYSQL_DEFAULT_AUTH, default_auth);
```

Typically, the program will also accept `--plugin-dir` and `--default-auth` options that enable users to override the default values.

17.1 mysql_client_find_plugin()

```
struct st_mysql_client_plugin *
mysql_client_find_plugin(MYSQL *mysql,
                         const char *name,
                         int type)
```

Description

Returns a pointer to a loaded plugin, loading the plugin first if necessary. An error occurs if the type is invalid or the plugin cannot be found or loaded.

Specify the arguments as follows:

- `mysql`: A pointer to a `MYSQL` structure. The plugin API does not require a connection to a MySQL server, but this structure must be properly initialized. The structure is used to obtain connection-related information.
- `name`: The plugin name.
- `type`: The plugin type.

Return Values

A pointer to the plugin for success. `NULL` if an error occurred.

Errors

To check for errors, call the `mysql_error()` or `mysql_errno()` function. See Section 7.16, “mysql_error()”, and Section 7.15, “mysql_errno()”.

Example

```
MYSQL mysql;
struct st_mysql_client_plugin *p;

if ((p = mysql_client_find_plugin(&mysql, "myplugin",
                                  MYSQL_CLIENT_AUTHENTICATION_PLUGIN, 0)))
{
  printf("Plugin version: %d.%d.%d\n", p->version[0], p->version[1], p->version[2]);
}
```

17.2 mysql_client_register_plugin()

```
struct st_mysql_client_plugin *
mysql_client_register_plugin(MYSQL *mysql,
                             struct st_mysql_client_plugin *plugin)
```

Description

Adds a plugin structure to the list of loaded plugins. An error occurs if the plugin is already loaded.

Specify the arguments as follows:

- `mysql`: A pointer to a `MYSQL` structure. The plugin API does not require a connection to a MySQL server, but this structure must be properly initialized. The structure is used to obtain connection-related information.
- `plugin`: A pointer to the plugin structure.

Return Values

A pointer to the plugin for success. `NULL` if an error occurred.

Errors

To check for errors, call the `mysql_error()` or `mysql_errno()` function. See Section 7.16, “mysql_error()”, and Section 7.15, “mysql_errno()”.

17.3 mysql_load_plugin()

```
struct st_mysql_client_plugin *
mysql_load_plugin(MYSQL *mysql,
                  const char *name,
                  int type,
                  int argc,
                  ...)
```

Description

Loads a MySQL client plugin, specified by name and type. An error occurs if the type is invalid or the plugin cannot be loaded.

It is not possible to load multiple plugins of the same type. An error occurs if you try to load a plugin of a type already loaded.

Specify the arguments as follows:

- `mysql`: A pointer to a `MYSQL` structure. The plugin API does not require a connection to a MySQL server, but this structure must be properly initialized. The structure is used to obtain connection-related information.
- `name`: The name of the plugin to load.
- `type`: The type of plugin to load, or `-1` to disable type checking. If type is not `-1`, only plugins matching the type are considered for loading.
- `argc`: The number of following arguments (0 if there are none). Interpretation of any following arguments depends on the plugin type.

Another way to cause plugins to be loaded is to set the `LIBMYSQL_PLUGINS` environment variable to a list of semicolon-separated plugin names. For example:

```
export LIBMYSQL_PLUGINS="myplugin1;myplugin2"
```

Plugins named by `LIBMYSQL_PLUGINS` are loaded when the client program calls `mysql_library_init()`. No error is reported if problems occur loading these plugins.

The `LIBMYSQL_PLUGIN_DIR` environment variable can be set to the path name of the directory in which to look for client plugins. This variable is used in two ways:

- During client plugin preloading, the value of the `--plugin-dir` option is not available, so client plugin loading fails unless the plugins are located in the hardwired default directory. If the plugins are located elsewhere, `LIBMYSQL_PLUGIN_DIR` environment variable can be set to the proper directory to enable plugin preloading to succeed.
- For explicit client plugin loading, the `mysql_load_plugin()` and `mysql_load_plugin_v()` C API functions use the `LIBMYSQL_PLUGIN_DIR` value if it exists and the `--plugin-dir` option was not given. If `--plugin-dir` is given, `mysql_load_plugin()` and `mysql_load_plugin_v()` ignore `LIBMYSQL_PLUGIN_DIR`.

Return Values

A pointer to the plugin if it was loaded successfully. `NULL` if an error occurred.

Errors

To check for errors, call the `mysql_error()` or `mysql_errno()` function. See Section 7.16, “mysql_error()”, and Section 7.15, “mysql_errno()”.

Example

```
MYSQL mysql;

if(!mysql_load_plugin(&mysql, "myplugin",
                      MYSQL_CLIENT_AUTHENTICATION_PLUGIN, 0))
{
    fprintf(stderr, "Error: %s\n", mysql_error(&mysql));
    exit(-1);
}
```

See Also

See also Section 17.3, “mysql_load_plugin()”, Section 7.16, “mysql_error()”, Section 7.15, “mysql_errno()”.

17.4 mysql_load_plugin_v()

```
struct st_mysql_client_plugin *
mysql_load_plugin_v(MYSQL *mysql,
                    const char *name,
                    int type,
                    int argc,
                    va_list args)
```

Description

This function is equivalent to `mysql_load_plugin()`, but it accepts a `va_list` instead of a variable list of arguments.

See Also

See also Section 17.3, “mysql_load_plugin()”.

17.5 mysql_plugin_options()

```
int
mysql_plugin_options(struct st_mysql_client_plugin *plugin,
                     const char *option,
                     const void *value)
```

Description

Passes an option type and value to a plugin. This function can be called multiple times to set several options. If the plugin does not have an option handler, an error occurs.

Specify the arguments as follows:

- `plugin`: A pointer to the plugin structure.
- `option`: The option to be set.
- `value`: A pointer to the option value.

Return Values

Zero for success, 1 if an error occurred. If the plugin has an option handler, that handler should also return zero for success and 1 if an error occurred.

Chapter 18 C API Binary Log Interface

The MySQL client/server protocol includes a client interface for reading a stream of replication events from a MySQL server binary log. This capability uses the `MYSQL_RPL` data structure and a small set of functions to manage communication between a client program and the server from which the binary log is to be read. The following sections describe aspects of this interface in more detail.

Chapter 19 C API Binary Log Data Structures

C API functions for processing a replication event stream from a server require a connection handler (a `MYSQL *` pointer) and a pointer to a `MYSQL_RPL` structure that describes the steam of replication events to read from the server binary log. For example:

```
MYSQL *mysql = mysql_real_connect(...);

MYSQL_RPL rpl;

# ... initialize MYSQL_RPL members ...

int result = mysql_binlog_open(mysql, &rpl);
```

This section describes the `MYSQL_RPL` structure members. Connection handlers are described in Chapter 5, *C API Data Structures*.

The applicable `MYSQL_RPL` members depend on the binary log operation to be performed:

- Before calling `mysql_binlog_open()`, the caller must set the `MYSQL_RPL` members from `file_name_length` through `flags`. In addition, if `flags` has the `MYSQL_RPL_GTID` flag set, the caller must set the members from `gtid_set_encoded_size` through `gtid_set_arg`.
- After a successful `mysql_binlog_fetch()` call, the caller examines the `size` and `buffer` members.

`MYSQL_RPL` structure member descriptions:

- `file_name_length`

  The length of the name of the binary log file to read. This member is used in conjunction with `file_name`; see the `file_name` description.

- `file_name`

  The name of the binary log file to read:

  - If `file_name` is `NULL`, the client library sets it to the empty string and sets `file_name_length` to 0.
  - If `file_name` is not `NULL`, `file_name_length` must either be the length of the name or 0. If `file_name_length` is 0, the client library sets it to the length of the name, in which case, `file_name` must be given as a null-terminated string.

  To read from the beginning of the binary log without having to know the name of the oldest binary log file, set `file_name` to `NULL` or the empty string, and `start_position` to 4.

- `start_position`

  The position at which to start reading the binary log. The position of the first event in any given binary log file is 4.

- `server_id`

  The server ID to use for identifying to the server from which the binary log is read.

- `flags`

  The union of flags that affect binary log reading, or 0 if no flags are set. These flag values are permitted:

  - `MYSQL_RPL_SKIP_HEARTBEAT`

Set this flag to cause `mysql_binlog_fetch()` to skip heartbeat events.

- `MYSQL_RPL_GTID`

  Set this flag to read GTID (global transaction ID) data. If set, you must initialize the `MYSQL_RPL` structure GTID-related members from `gtid_set_encoded_size` to `gtid_set_arg` before calling `mysql_binlog_open()`.

  It is beyond the scope of this documentation to describe in detail how client programs use those GTID-related members. For more information, examine the `mysqlbinlog.cc` source file. For information about GTID-based replication, see Replication with Global Transaction Identifiers.

- `gtid_set_encoded_size`

  The size of GTID set data, or 0.

- `fix_gtid_set`

  The address of a callback function for `mysql_binlog_open()` to call to fill the command packet GTID set, or `NULL` if there is no such function. The callback function, if used, should have this calling signature:

  ```
  void my_callback(MYSQL_RPL *rpl, unsigned char *packet_gtid_set);
  ```

- `gtid_set_arg`

  Either a pointer to GTID set data (if `fix_gtid_set` is `NULL`), or a pointer to a value to be made available for use within the callback function (if `fix_gtid_set` is not `NULL`). `gtid_set_arg` is a generic pointer, so it can point to any kind of value (for example, a string, a structure, or a function). Its interpretation within the callback depends on how the callback intends to use it.

- `size`

  After a successful `mysql_binlog_fetch()` call, the size of the returned binary log event. The value is 0 for an EOF event, greater than 0 for a non-EOF event.

- `buffer`

  After a successful `mysql_binlog_fetch()` call, a pointer to the binary log event contents.

Chapter 20 C API Binary Log Function Overview

The following list summarizes the functions available for reading a replication event stream from a binary log. For greater detail, see the descriptions in Chapter 21, *C API Binary Log Function Descriptions*.

- `mysql_binlog_close()`: Closes the replication event stream.
- `mysql_binlog_fetch()`: Reads an event from the replication event stream.
- `mysql_binlog_open()`: Opens the replication event stream.

Chapter 21 C API Binary Log Function Descriptions

Table of Contents

The following sections provide detailed descriptions of the functions that enable reading the stream of replication events from a MySQL server binary log.

The following simple example program demonstrates the binary log C API functions. Program notes:

- `mysql` is assumed to be a valid connection handler.
- The initial `SET` statement sets the `@source_binlog_checksum` user-defined variable that the server takes as an indication that the client is checksum-aware. This client does nothing with checksums, but without this statement, a server that includes checksums in binary log events will return an error for the first attempt to read an event containing a checksum. The value assigned to the variable is immaterial; what matters is that the variable exist. existence.

```
if (mysql_query(mysql, "SET @source_binlog_checksum='ALL'"))
{
  fprintf(stderr, "mysql_query() failed\n");
  fprintf(stderr, "Error %u: %s\n",
           mysql_errno(mysql), mysql_error(mysql));
  exit(1);
}

MYSQL_RPL rpl;

rpl.file_name_length = 0;
rpl.file_name = NULL;
rpl.start_position = 4;
rpl.server_id = 0;
rpl.flags = 0;

if (mysql_binlog_open(mysql, &rpl))
{
  fprintf(stderr, "mysql_binlog_open() failed\n");
  fprintf(stderr, "Error %u: %s\n",
           mysql_errno(mysql), mysql_error(mysql));
  exit(1);
}
for (;;)  /* read events until error or EOF */
{
  if (mysql_binlog_fetch(mysql, &rpl))
  {
    fprintf(stderr, "mysql_binlog_fetch() failed\n");
    fprintf(stderr, "Error %u: %s\n",
             mysql_errno(mysql), mysql_error(mysql));
    break;
  }
  if (rpl.size == 0)  /* EOF */
  {
    fprintf(stderr, "EOF event received\n");
    break;
  }
  fprintf(stderr, "Event received of size %lu.\n", rpl.size);
}
```

```
mysql_binlog_close(mysql, &rpl);
```

For additional examples that show how to use these functions, look in a MySQL source distribution for these source files:

- `mysqlbinlog.cc` in the `client` directory
- `mysql_client_test.c` in the `testclients` directory

21.1 mysql_binlog_close()

```
void
mysql_binlog_close(MYSQL *mysql,
                   MYSQL_RPL *rpl)
```

Description

Close a replication event stream.

Arguments:

- `mysql`: The connection handler returned from `mysql_init()`. The handler remains open after the `mysql_binlog_close()` call.
- `rpl`: The replication stream structure. After calling `mysql_binlog_close()`, this structure should not be used further without reinitializing it and calling `mysql_binlog_open()` again.

Errors

None.

Example

See Chapter 21, *C API Binary Log Function Descriptions*.

21.2 mysql_binlog_fetch()

```
int
mysql_binlog_fetch(MYSQL *mysql,
                   MYSQL_RPL *rpl)
```

Description

Fetch one event from the replication event stream.

Arguments:

- `mysql`: The connection handler returned from `mysql_init()`.
- `rpl`: The replication stream structure. After a successful call, the `size` member indicates the event size, which is 0 for an EOF event. For a non-EOF event, `size` is greater than 0 and the `buffer` member points to the event contents.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

Example

See Chapter 21, *C API Binary Log Function Descriptions*.

21.3 mysql_binlog_open()

```
int
mysql_binlog_open(MYSQL *mysql,
                  MYSQL_RPL *rpl)
```

Description

Open a new replication event stream, to read a MySQL server binary log.

Arguments:

- `mysql`: The connection handler returned from `mysql_init()`.
- `rpl`: A `MYSQL_RPL` structure that has been initialized to indicate the replication event stream source. For a description of the structure members and how to initialize them, see Chapter 19, *C API Binary Log Data Structures*.

Return Values

Zero for success. Nonzero if an error occurred.

Errors

- `CR_FILE_NAME_TOO_LONG`

  The specified binary log file name was too long.

- `CR_OUT_OF_MEMORY`

  Out of memory.

Example

See Chapter 21, *C API Binary Log Function Descriptions*.

Chapter 22 C API Support for Encrypted Connections

This section describes how C applications use the C API capabilities for encrypted connections. By default, MySQL programs attempt to connect using encryption if the server supports encrypted connections, falling back to an unencrypted connection if an encrypted connection cannot be established (see Configuring MySQL to Use Encrypted Connections). For applications that require control beyond the default behavior over how encrypted connections are established, the C API provides these capabilities:

- The `mysql_options()` function enables applications to set the appropriate SSL/TLS options before calling `mysql_real_connect()`. For example, to require the use of an encrypted connection, see Enforcing an Encrypted Connection.
- The `mysql_get_ssl_cipher()` function enables applications to determine, after a connection has been established, whether the connection uses encryption. A `NULL` return value indicates that encryption is not being used. A non-`NULL` return value indicates an encrypted connection and names the encryption cipher. See Section 7.35, “mysql_get_ssl_cipher()”.
- C API Options for Encrypted Connections
- Enforcing an Encrypted Connection
- Improving Security of Encrypted Connections

C API Options for Encrypted Connections

`mysql_options()` provides the following options for control over use of encrypted connections. For option details, see Section 7.51, “mysql_options()”.

- `MYSQL_OPT_SSL_CA`: The path name of the Certificate Authority (CA) certificate file. This option, if used, must specify the same certificate used by the server.
- `MYSQL_OPT_SSL_CAPATH`: The path name of the directory that contains trusted SSL CA certificate files.
- `MYSQL_OPT_SSL_CERT`: The path name of the client public key certificate file.
- `MYSQL_OPT_SSL_CIPHER`: The list of encryption ciphers the client permits for connections that use TLS protocols up through TLSv1.2.
- `MYSQL_OPT_SSL_CRL`: The path name of the file containing certificate revocation lists.
- `MYSQL_OPT_SSL_CRLPATH`: The path name of the directory that contains certificate revocation list files.
- `MYSQL_OPT_SSL_KEY`: The path name of the client private key file.
- `MYSQL_OPT_SSL_MODE`: The connection security state.
- `MYSQL_OPT_TLS_CIPHERSUITES`: The list of encryption ciphersuites the client permits for connections that use TLSv1.3.
- `MYSQL_OPT_TLS_VERSION`: The encryption protocols the client permits.

`mysql_ssl_set()` can be used as a convenience routine that is equivalent to a set of `mysql_options()` calls that specify certificate and key files, encryption ciphers, and so forth. See Section 7.79, “mysql_ssl_set()”.

Enforcing an Encrypted Connection

`mysql_options()` options for information such as SSL certificate and key files are used to establish an encrypted connection if such connections are available, but do not enforce any requirement that the connection obtained be encrypted. To require an encrypted connection, use the following technique:

1. Call `mysql_options()` as necessary supply the appropriate SSL parameters (certificate and key files, encryption ciphers, and so forth).
2. Call `mysql_options()` to pass the `MYSQL_OPT_SSL_MODE` option with a value of `SSL_MODE_REQUIRED` or one of the more-restrictive option values.
3. Call `mysql_real_connect()` to connect to the server. The call fails if an encrypted connection cannot be obtained; exit with an error.

Improving Security of Encrypted Connections

For additional security relative to that provided by the default encryption, clients can supply a CA certificate matching the one used by the server and enable host name identity verification. In this way, the server and client place their trust in the same CA certificate and the client verifies that the host to which it connected is the one intended:

- To specify the CA certificate, call `mysql_options()` to pass the `MYSQL_OPT_SSL_CA` (or `MYSQL_OPT_SSL_CAPATH`) option, and call `mysql_options()` to pass the `MYSQL_OPT_SSL_MODE` option with a value of `SSL_MODE_VERIFY_CA`.
- To enable host name identity verification as well, call `mysql_options()` to pass the `MYSQL_OPT_SSL_MODE` option with a value of `SSL_MODE_VERIFY_IDENTITY` rather than `SSL_MODE_VERIFY_CA`.

> **Note**
>
> Host name identity verification with `SSL_MODE_VERIFY_IDENTITY` does not work with self-signed certificates created automatically by the server, or manually using `mysql_ssl_rsa_setup` (see Creating SSL and RSA Certificates and Keys using MySQL). Such self-signed certificates do not contain the server name as the Common Name value.
>
> Host name identity verification also does not work with certificates that specify the Common Name using wildcards because that name is compared verbatim to the server name.

Chapter 23 C API Multiple Statement Execution Support

By default, `mysql_query()` and `mysql_real_query()` interpret their statement string argument as a single statement to be executed, and you process the result according to whether the statement produces a result set (a set of rows, as for `SELECT`) or an affected-rows count (as for `INSERT`, `UPDATE`, and so forth).

MySQL also supports the execution of a string containing multiple statements separated by semicolon (`;`) characters. This capability is enabled by special options that are specified either when you connect to the server with `mysql_real_connect()` or after connecting by calling `mysql_set_server_option()`.

Executing a multiple-statement string can produce multiple result sets or row-count indicators. Processing these results involves a different approach than for the single-statement case: After handling the result from the first statement, it is necessary to check whether more results exist and process them in turn if so. To support multiple-result processing, the C API includes the `mysql_more_results()` and `mysql_next_result()` functions. These functions are used at the end of a loop that iterates as long as more results are available. *Failure to process the result this way may result in a dropped connection to the server.*

Multiple-result processing also is required if you execute `CALL` statements for stored procedures. Results from a stored procedure have these characteristics:

- Statements within the procedure may produce result sets (for example, if it executes `SELECT` statements). These result sets are returned in the order that they are produced as the procedure executes.

  In general, the caller cannot know how many result sets a procedure will return. Procedure execution may depend on loops or conditional statements that cause the execution path to differ from one call to the next. Therefore, you must be prepared to retrieve multiple results.

- The final result from the procedure is a status result that includes no result set. The status indicates whether the procedure succeeded or an error occurred.

The multiple statement and result capabilities can be used only with `mysql_query()` or `mysql_real_query()`. They cannot be used with the prepared statement interface. Prepared statement handlers are defined to work only with strings that contain a single statement. See Chapter 8, *C API Prepared Statements*.

To enable multiple-statement execution and result processing, the following options may be used:

- The `mysql_real_connect()` function has a `flags` argument for which two option values are relevant:

  - `CLIENT_MULTI_RESULTS` enables the client program to process multiple results. This option *must* be enabled if you execute `CALL` statements for stored procedures that produce result sets. Otherwise, such procedures result in an error `Error 1312 (0A000): PROCEDURE` *`proc_name`* `can't return a result set in the given context`. `CLIENT_MULTI_RESULTS` is enabled by default.

  - `CLIENT_MULTI_STATEMENTS` enables `mysql_query()` and `mysql_real_query()` to execute statement strings containing multiple statements separated by semicolons. This option also enables `CLIENT_MULTI_RESULTS` implicitly, so a `flags` argument of `CLIENT_MULTI_STATEMENTS` to `mysql_real_connect()` is equivalent to an argument of `CLIENT_MULTI_STATEMENTS | CLIENT_MULTI_RESULTS`. That is, `CLIENT_MULTI_STATEMENTS` is sufficient to enable multiple-statement execution and all multiple-result processing.

- After the connection to the server has been established, you can use the `mysql_set_server_option()` function to enable or disable multiple-statement execution by passing it an argument of `MYSQL_OPTION_MULTI_STATEMENTS_ON` or `MYSQL_OPTION_MULTI_STATEMENTS_OFF`. Enabling multiple-statement execution with this function also enables processing of “simple” results for a multiple-statement string where each statement produces a single result, but is *not* sufficient to permit processing of stored procedures that produce result sets.

The following procedure outlines a suggested strategy for handling multiple statements:

1. Pass `CLIENT_MULTI_STATEMENTS` to `mysql_real_connect()`, to fully enable multiple-statement execution and multiple-result processing.
2. After calling `mysql_query()` or `mysql_real_query()` and verifying that it succeeds, enter a loop within which you process statement results.
3. For each iteration of the loop, handle the current statement result, retrieving either a result set or an affected-rows count. If an error occurs, exit the loop.
4. At the end of the loop, call `mysql_next_result()` to check whether another result exists and initiate retrieval for it if so. If no more results are available, exit the loop.

One possible implementation of the preceding strategy is shown following. The final part of the loop can be reduced to a simple test of whether `mysql_next_result()` returns nonzero. The code as written distinguishes between no more results and an error, which enables a message to be printed for the latter occurrence.

```
/* connect to server with the CLIENT_MULTI_STATEMENTS option */
if (mysql_real_connect (mysql, host_name, user_name, password,
    db_name, port_num, socket_name, CLIENT_MULTI_STATEMENTS) == NULL)
{
  printf("mysql_real_connect() failed\n");
  mysql_close(mysql);
  exit(1);
}

/* execute multiple statements */
status = mysql_query(mysql,
                     "DROP TABLE IF EXISTS test_table;\
                      CREATE TABLE test_table(id INT);\
                      INSERT INTO test_table VALUES(10);\
                      UPDATE test_table SET id=20 WHERE id=10;\
                      SELECT * FROM test_table;\
                      DROP TABLE test_table");
if (status)
{
  printf("Could not execute statement(s)");
  mysql_close(mysql);
  exit(0);
}

/* process each statement result */
do {
  /* did current statement return data? */
  result = mysql_store_result(mysql);
  if (result)
  {
    /* yes; process rows and free the result set */
    process_result_set(mysql, result);
    mysql_free_result(result);
  }
  else          /* no result set or error */
  {
```

```
    if (mysql_field_count(mysql) == 0)
    {
      printf("%lld rows affected\n",
            mysql_affected_rows(mysql));
    }
    else  /* some error occurred */
    {
      printf("Could not retrieve result set\n");
      break;
    }
  }
  /* more results? -1 = no, >0 = error, 0 = yes (keep looping) */
  if ((status = mysql_next_result(mysql)) > 0)
    printf("Could not execute statement\n");
} while (status == 0);

mysql_close(mysql);
```

Chapter 24 C API Prepared Statement Handling of Date and Time Values

The binary (prepared statement) protocol enables you to send and receive date and time values (DATE, TIME, DATETIME, and TIMESTAMP), using the MYSQL_TIME structure. The members of this structure are described in Chapter 9, *C API Prepared Statement Data Structures*.

To send temporal data values, create a prepared statement using mysql_stmt_prepare(). Then, before calling mysql_stmt_execute() to execute the statement, use the following procedure to set up each temporal parameter:

1. In the MYSQL_BIND structure associated with the data value, set the buffer_type member to the type that indicates what kind of temporal value you're sending. For DATE, TIME, DATETIME, or TIMESTAMP values, set buffer_type to MYSQL_TYPE_DATE, MYSQL_TYPE_TIME, MYSQL_TYPE_DATETIME, or MYSQL_TYPE_TIMESTAMP, respectively.

2. Set the buffer member of the MYSQL_BIND structure to the address of the MYSQL_TIME structure in which you pass the temporal value.

3. Fill in the members of the MYSQL_TIME structure that are appropriate for the type of temporal value to pass.

Use mysql_stmt_bind_param() to bind the parameter data to the statement. Then you can call mysql_stmt_execute().

To retrieve temporal values, the procedure is similar, except that you set the buffer_type member to the type of value you expect to receive, and the buffer member to the address of a MYSQL_TIME structure into which the returned value should be placed. Use mysql_stmt_bind_result() to bind the buffers to the statement after calling mysql_stmt_execute() and before fetching the results.

Here is a simple example that inserts DATE, TIME, and TIMESTAMP data. The mysql variable is assumed to be a valid connection handler.

```
MYSQL_TIME  ts;
MYSQL_BIND  bind[3];
MYSQL_STMT  *stmt;

strmov(query, "INSERT INTO test_table(date_field, time_field, \
                              timestamp_field) VALUES(?,?,?");

stmt = mysql_stmt_init(mysql);
if (!stmt)
{
  fprintf(stderr, " mysql_stmt_init(), out of memory\n");
  exit(0);
}
if (mysql_stmt_prepare(mysql, query, strlen(query)))
{
  fprintf(stderr, "\n mysql_stmt_prepare(), INSERT failed");
  fprintf(stderr, "\n %s", mysql_stmt_error(stmt));
  exit(0);
}

/* set up input buffers for all 3 parameters */
bind[0].buffer_type= MYSQL_TYPE_DATE;
bind[0].buffer= (char *)&ts;
bind[0].is_null= 0;
bind[0].length= 0;
```

```
...
bind[1]= bind[2]= bind[0];
...

mysql_stmt_bind_param(stmt, bind);

/* supply the data to be sent in the ts structure */
ts.year= 2002;
ts.month= 02;
ts.day= 03;

ts.hour= 10;
ts.minute= 45;
ts.second= 20;

mysql_stmt_execute(stmt);
..
```

Chapter 25 C API Prepared CALL Statement Support

This section describes prepared-statement support in the C API for stored procedures executed using `CALL` statements:

Stored procedures executed using prepared `CALL` statements can be used in the following ways:

- A stored procedure can produce any number of result sets. The number of columns and the data types of the columns need not be the same for all result sets.
- The final values of `OUT` and `INOUT` parameters are available to the calling application after the procedure returns. These parameters are returned as an extra single-row result set following any result sets produced by the procedure itself. The row contains the values of the `OUT` and `INOUT` parameters in the order in which they are declared in the procedure parameter list.

  For information about the effect of unhandled conditions on procedure parameters, see Condition Handling and OUT or INOUT Parameters.

The following discussion shows how to use these capabilities through the C API for prepared statements. To use prepared `CALL` statements through the `PREPARE` and `EXECUTE` statements, see CALL Statement.

An application that executes a prepared `CALL` statement should use a loop that fetches a result and then invokes `mysql_stmt_next_result()` to determine whether there are more results. The results consist of any result sets produced by the stored procedure followed by a final status value that indicates whether the procedure terminated successfully.

If the procedure has `OUT` or `INOUT` parameters, the result set preceding the final status value contains their values. To determine whether a result set contains parameter values, test whether the `SERVER_PS_OUT_PARAMS` bit is set in the `server_status` member of the `MYSQL` connection handler:

```
mysql->server_status & SERVER_PS_OUT_PARAMS
```

The following example uses a prepared `CALL` statement to execute a stored procedure that produces multiple result sets and that provides parameter values back to the caller by means of `OUT` and `INOUT` parameters. The procedure takes parameters of all three types (`IN`, `OUT`, `INOUT`), displays their initial values, assigns new values, displays the updated values, and returns. The expected return information from the procedure therefore consists of multiple result sets and a final status:

- One result set from a `SELECT` that displays the initial parameter values: `10`, `NULL`, `30`. (The `OUT` parameter is assigned a value by the caller, but this assignment is expected to be ineffective: `OUT` parameters are seen as `NULL` within a procedure until assigned a value within the procedure.)
- One result set from a `SELECT` that displays the modified parameter values: `100`, `200`, `300`.
- One result set containing the final `OUT` and `INOUT` parameter values: `200`, `300`.
- A final status packet.

The code to execute the procedure:

```
MYSQL_STMT *stmt;
MYSQL_BIND ps_params[3];  /* input parameter buffers */
int        int_data[3];   /* input/output values */
bool       is_null[3];    /* output value nullability */
int        status;

/* set up stored procedure */
status = mysql_query(mysql, "DROP PROCEDURE IF EXISTS p1");
test_error(mysql, status);
```

```
status = mysql_query(mysql,
  "CREATE PROCEDURE p1("
  "  IN p_in INT, "
  "  OUT p_out INT, "
  "  INOUT p_inout INT) "
  "BEGIN "
  "  SELECT p_in, p_out, p_inout; "
  "  SET p_in = 100, p_out = 200, p_inout = 300; "
  "  SELECT p_in, p_out, p_inout; "
  "END");
test_error(mysql, status);

/* initialize and prepare CALL statement with parameter placeholders */
stmt = mysql_stmt_init(mysql);
if (!stmt)
{
  printf("Could not initialize statement\n");
  exit(1);
}
status = mysql_stmt_prepare(stmt, "CALL p1(?, ?, ?)", 16);
test_stmt_error(stmt, status);

/* initialize parameters: p_in, p_out, p_inout (all INT) */
memset(ps_params, 0, sizeof (ps_params));

ps_params[0].buffer_type = MYSQL_TYPE_LONG;
ps_params[0].buffer = (char *) &int_data[0];
ps_params[0].length = 0;
ps_params[0].is_null = 0;

ps_params[1].buffer_type = MYSQL_TYPE_LONG;
ps_params[1].buffer = (char *) &int_data[1];
ps_params[1].length = 0;
ps_params[1].is_null = 0;

ps_params[2].buffer_type = MYSQL_TYPE_LONG;
ps_params[2].buffer = (char *) &int_data[2];
ps_params[2].length = 0;
ps_params[2].is_null = 0;

/* bind parameters */
status = mysql_stmt_bind_param(stmt, ps_params);
test_stmt_error(stmt, status);

/* assign values to parameters and execute statement */
int_data[0]= 10;  /* p_in */
int_data[1]= 20;  /* p_out */
int_data[2]= 30;  /* p_inout */

status = mysql_stmt_execute(stmt);
test_stmt_error(stmt, status);

/* process results until there are no more */
do {
  int i;
  int num_fields;       /* number of columns in result */
  MYSQL_FIELD *fields;  /* for result set metadata */
  MYSQL_BIND *rs_bind;  /* for output buffers */

  /* the column count is > 0 if there is a result set */
  /* 0 if the result is only the final status packet */
  num_fields = mysql_stmt_field_count(stmt);

  if (num_fields > 0)
  {
    /* there is a result set to fetch */
    printf("Number of columns in result: %d\n", (int) num_fields);
```

```
/* what kind of result set is this? */
printf("Data: ");
if(mysql->server_status & SERVER_PS_OUT_PARAMS)
  printf("this result set contains OUT/INOUT parameters\n");
else
  printf("this result set is produced by the procedure\n");

MYSQL_RES *rs_metadata = mysql_stmt_result_metadata(stmt);
test_stmt_error(stmt, rs_metadata == NULL);

fields = mysql_fetch_fields(rs_metadata);

rs_bind = (MYSQL_BIND *) malloc(sizeof (MYSQL_BIND) * num_fields);
if (!rs_bind)
{
  printf("Cannot allocate output buffers\n");
  exit(1);
}
memset(rs_bind, 0, sizeof (MYSQL_BIND) * num_fields);

/* set up and bind result set output buffers */
for (i = 0; i < num_fields; ++i)
{
  rs_bind[i].buffer_type = fields[i].type;
  rs_bind[i].is_null = &is_null[i];

  switch (fields[i].type)
  {
    case MYSQL_TYPE_LONG:
      rs_bind[i].buffer = (char *) &(int_data[i]);
      rs_bind[i].buffer_length = sizeof (int_data);
      break;

    default:
      fprintf(stderr, "ERROR: unexpected type: %d.\n", fields[i].type);
      exit(1);
  }
}

status = mysql_stmt_bind_result(stmt, rs_bind);
test_stmt_error(stmt, status);

/* fetch and display result set rows */
while (1)
{
  status = mysql_stmt_fetch(stmt);

  if (status == 1 || status == MYSQL_NO_DATA)
    break;

  for (i = 0; i < num_fields; ++i)
  {
    switch (rs_bind[i].buffer_type)
    {
      case MYSQL_TYPE_LONG:
        if (*rs_bind[i].is_null)
          printf(" val[%d] = NULL;", i);
        else
          printf(" val[%d] = %ld;",
                 i, (long) *((int *) rs_bind[i].buffer));
        break;

      default:
        printf("  unexpected type (%d)\n",
          rs_bind[i].buffer_type);
    }
```

```
      }
      printf("\n");
    }

    mysql_free_result(rs_metadata); /* free metadata */
    free(rs_bind);                  /* free output buffers */
  }
  else
  {
    /* no columns = final status packet */
    printf("End of procedure output\n");
  }

  /* more results? -1 = no, >0 = error, 0 = yes (keep looking) */
  status = mysql_stmt_next_result(stmt);
  if (status > 0)
    test_stmt_error(stmt, status);
} while (status == 0);

mysql_stmt_close(stmt);
```

Execution of the procedure should produce the following output:

```
Number of columns in result: 3
Data: this result set is produced by the procedure
 val[0] = 10; val[1] = NULL; val[2] = 30;
Number of columns in result: 3
Data: this result set is produced by the procedure
 val[0] = 100; val[1] = 200; val[2] = 300;
Number of columns in result: 2
Data: this result set contains OUT/INOUT parameters
 val[0] = 200; val[1] = 300;
End of procedure output
```

The code uses two utility routines, `test_error()` and `test_stmt_error()`, to check for errors and terminate after printing diagnostic information if an error occurred:

```
static void test_error(MYSQL *mysql, int status)
{
  if (status)
  {
    fprintf(stderr, "Error: %s (errno: %d)\n",
            mysql_error(mysql), mysql_errno(mysql));
    exit(1);
  }
}

static void test_stmt_error(MYSQL_STMT *stmt, int status)
{
  if (status)
  {
    fprintf(stderr, "Error: %s (errno: %d)\n",
            mysql_stmt_error(stmt), mysql_stmt_errno(stmt));
    exit(1);
  }
}
```

Chapter 26 C API Prepared Statement Problems

Here follows a list of the currently known problems with prepared statements:

- `TIME`, `TIMESTAMP`, and `DATETIME` do not support parts of seconds (for example, from `DATE_FORMAT()`).
- When converting an integer to string, `ZEROFILL` is honored with prepared statements in some cases where the MySQL server does not print the leading zeros. (For example, with `MIN(number-with-zerofill)`).
- When converting a floating-point number to a string in the client, the rightmost digits of the converted value may differ slightly from those of the original value.
- Prepared statements do not support multi-statements (that is, multiple statements within a single string separated by `;` characters).
- The capabilities of prepared `CALL` statements are described in Chapter 25, *C API Prepared CALL Statement Support*.

Chapter 27 C API Optional Result Set Metadata

When a client executes a statement that produces a result set, MySQL makes available the data the result set contains, and by default also result set metadata that provides information about the result set data. Metadata is contained in the `MYSQL_FIELD` structure (see Chapter 5, *C API Data Structures*), which is returned by the `mysql_fetch_field()`, `mysql_fetch_field_direct()`, and `mysql_fetch_fields()` functions.

Clients can indicate on a per-connection basis that result set metadata is optional and that the client will indicate to the server whether to return it. Suppression of metadata transfer by the client can improve performance, particularly for sessions that execute many queries that return few rows each.

There are two ways for a client to indicate that result set metadata is optional for a connection. They are equivalent, so either one suffices:

- Prior to connect time, enable the `MYSQL_OPT_OPTIONAL_RESULTSET_METADATA` option for `mysql_options()`.
- At connect time, enable the `CLIENT_OPTIONAL_RESULTSET_METADATA` flag for the `client_flag` argument of `mysql_real_connect()`.

For metadata-optional connections, the client sets the `resultset_metadata` system variable to control whether the server returns result set metadata. Permitted values are `FULL` (return all metadata) and `NONE` (return no metadata). The default is `FULL`, so even for metadata-optional connections, the server by default returns metadata.

For metadata-optional connections, the `mysql_fetch_field()`, `mysql_fetch_field_direct()`, and `mysql_fetch_fields()` functions return `NULL` when `resultset_metadata` is set to `NONE`.

For connections that are not metadata-optional, setting `resultset_metadata` to `NONE` produces an error.

To check whether a result set has metadata, the client calls the `mysql_result_metadata()` function. This function returns `RESULTSET_METADATA_FULL` or `RESULTSET_METADATA_NONE` to indicate that the result set has full metadata or no metadata, respectively.

`mysql_result_metadata()` is useful if the client does not know in advance whether a result set has metadata. For example, if a client executes a stored procedure that returns multiple result sets and might change the `resultset_metadata` system variable, the client can invoke `mysql_result_metadata()` for each result set to determine whether it has metadata.

Chapter 28 C API Automatic Reconnection Control

The MySQL client library can perform an automatic reconnection to the server if it finds that the connection is down when you attempt to send a statement to the server to be executed. If auto-reconnect is enabled, the library tries once to reconnect to the server and send the statement again.

Auto-reconnect is disabled by default.

If it is important for your application to know that the connection has been dropped (so that it can exit or take action to adjust for the loss of state information), be sure that auto-reconnect is disabled. To ensure this, call `mysql_options()` with the `MYSQL_OPT_RECONNECT` option:

```
bool reconnect = 0;
mysql_options(&mysql, MYSQL_OPT_RECONNECT, &reconnect);
```

If the connection has gone down, the effect of `mysql_ping()` depends on the auto-reconnect state. If auto-reconnect is enabled, `mysql_ping()` performs a reconnect. Otherwise, it returns an error.

Some client programs might provide the capability of controlling automatic reconnection. For example, `mysql` reconnects by default, but the `--skip-reconnect` option can be used to suppress this behavior.

If an automatic reconnection does occur (for example, as a result of calling `mysql_ping()`), there is no explicit indication of it. To check for reconnection, call `mysql_thread_id()` to get the original connection identifier before calling `mysql_ping()`, then call `mysql_thread_id()` again to see whether the identifier changed.

Automatic reconnection can be convenient because you need not implement your own reconnect code, but if a reconnection does occur, several aspects of the connection state are reset on the server side and your application will not be notified.

Reconnection affects the connection-related state as follows:

- Rolls back any active transactions and resets autocommit mode.
- Releases all table locks.
- Closes (and drops) all `TEMPORARY` tables.
- Reinitializes session system variables to the values of the corresponding global system variables, including system variables that are set implicitly by statements such as `SET NAMES`.
- Loses user-defined variable settings.
- Releases prepared statements.
- Closes `HANDLER` variables.
- Resets the value of `LAST_INSERT_ID()` to 0.
- Releases locks acquired with `GET_LOCK()`.
- Loses the association of the client with the Performance Schema `threads` table row that determines connection thread instrumentation. If the client reconnects after a disconnect, the session is associated with a new row in the `threads` table and the thread monitoring state may be different. See The threads Table.

If reconnection occurs, any SQL statement specified by calling `mysql_options()` with the `MYSQL_INIT_COMMAND` option is re-executed.

If the connection drops, it is possible that the session associated with the connection on the server side will still be running if the server has not yet detected that the client is no longer connected. In this case, any locks held by the original connection still belong to that session, so you may want to kill it by calling `mysql_kill()`.

Chapter 29 C API Common Issues

Table of Contents

29.1 Why mysql_store_result() Sometimes Returns NULL After mysql_query() Returns Success

It is possible for `mysql_store_result()` to return `NULL` following a successful call to `mysql_query()`. When this happens, it means one of the following conditions occurred:

- There was a `malloc()` failure (for example, if the result set was too large).
- The data could not be read (an error occurred on the connection).
- The query returned no data (for example, it was an `INSERT`, `UPDATE`, or `DELETE`).

You can always check whether the statement should have produced a nonempty result by calling `mysql_field_count()`. If `mysql_field_count()` returns zero, the result is empty and the last query was a statement that does not return values (for example, an `INSERT` or a `DELETE`). If `mysql_field_count()` returns a nonzero value, the statement should have produced a nonempty result. See the description of the `mysql_field_count()` function for an example.

You can test for an error by calling `mysql_error()` or `mysql_errno()`.

29.2 What Results You Can Get from a Query

In addition to the result set returned by a query, you can also get the following information:

- `mysql_affected_rows()` returns the number of rows affected by the last query when doing an `INSERT`, `UPDATE`, or `DELETE`.

  For a fast re-create, use `TRUNCATE TABLE`.

- `mysql_num_rows()` returns the number of rows in a result set. With `mysql_store_result()`, `mysql_num_rows()` may be called as soon as `mysql_store_result()` returns. With `mysql_use_result()`, `mysql_num_rows()` may be called only after you have fetched all the rows with `mysql_fetch_row()`.

- `mysql_insert_id()` returns the ID generated by the last query that inserted a row into a table with an `AUTO_INCREMENT` index. See Section 7.39, “mysql_insert_id()”.

- Some queries (`LOAD DATA`, `INSERT INTO ... SELECT`, `UPDATE`) return additional information. The result is returned by `mysql_info()`. See the description for `mysql_info()` for the format of the string that it returns. `mysql_info()` returns a `NULL` pointer if there is no additional information.

29.3 How to Get the Unique ID for the Last Inserted Row

If you insert a record into a table that contains an `AUTO_INCREMENT` column, you can obtain the value stored into that column by calling the `mysql_insert_id()` function.

You can check from your C applications whether a value was stored in an `AUTO_INCREMENT` column by executing the following code (which assumes that you've checked that the statement succeeded). It determines whether the query was an `INSERT` with an `AUTO_INCREMENT` index:

```
if ((result = mysql_store_result(&mysql)) == 0 &&
    mysql_field_count(&mysql) == 0 &&
    mysql_insert_id(&mysql) != 0)
{
    used_id = mysql_insert_id(&mysql);
}
```

When a new `AUTO_INCREMENT` value has been generated, you can also obtain it by executing a `SELECT LAST_INSERT_ID()` statement with `mysql_query()` and retrieving the value from the result set returned by the statement.

When inserting multiple values, the last automatically incremented value is returned.

For `LAST_INSERT_ID()`, the most recently generated ID is maintained in the server on a per-connection basis. It is not changed by another client. It is not even changed if you update another `AUTO_INCREMENT` column with a nonmagic value (that is, a value that is not `NULL` and not `0`). Using `LAST_INSERT_ID()` and `AUTO_INCREMENT` columns simultaneously from multiple clients is perfectly valid. Each client will receive the last inserted ID for the last statement *that* client executed.

If you want to use the ID that was generated for one table and insert it into a second table, you can use SQL statements like this:

```
INSERT INTO foo (auto,text)
    VALUES(NULL,'text');         # generate ID by inserting NULL
INSERT INTO foo2 (id,text)
    VALUES(LAST_INSERT_ID(),'text');  # use ID in second table
```

`mysql_insert_id()` returns the value stored into an `AUTO_INCREMENT` column, whether that value is automatically generated by storing `NULL` or `0` or was specified as an explicit value. `LAST_INSERT_ID()` returns only automatically generated `AUTO_INCREMENT` values. If you store an explicit value other than `NULL` or `0`, it does not affect the value returned by `LAST_INSERT_ID()`.

For more information on obtaining the last ID in an `AUTO_INCREMENT` column:

- For information on `LAST_INSERT_ID()`, which can be used within an SQL statement, see Information Functions.
- For information on `mysql_insert_id()`, the function you use from within the C API, see Section 7.39, “mysql_insert_id()”.
- For information on obtaining the auto-incremented value when using Connector/J, see Retrieving `AUTO_INCREMENT` Column Values through JDBC.
- For information on obtaining the auto-incremented value when using Connector/ODBC, see Obtaining Auto-Increment Values.

Index

Symbols

A

B

C

D

E

F

I

L

M

P

Q

R

S

T

U

W

Z