Extending MySQL 5.7

Abstract

This document describes what you need to know when working on the MySQL 5.7 code. To track or contribute to MySQL development, follow the instructions in Installing MySQL Using a Development Source Tree. If you are interested in MySQL internals, you should also join the MySQL Community Slack. Feel free to ask questions about the code and to send patches that you would like to contribute to the MySQL project!

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

Document generated on: 2021-08-05 (revision: 70529)

Table of Contents

Preface and Legal Notices

This document describes what you need to know when working on the MySQL 5.7 code. To track or contribute to MySQL development, follow the instructions in Installing MySQL Using a Development Source Tree. If you are interested in MySQL internals, you should also join the MySQL Community Slack. Feel free to ask questions about the code and to send patches that you would like to contribute to the MySQL project!

Legal Notices

expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

This documentation is NOT distributed under a GPL license. Use of this documentation is subject to the following terms:

You may create a printed copy of this documentation solely for your own personal use. Conversion to other formats is allowed as long as the actual content is not altered or edited in any way. You shall not publish or distribute this documentation in any form or on any media, except if you distribute the documentation in a manner similar to how Oracle disseminates it (that is, electronically for download on a Web site with the software) or on a CD-ROM or similar medium, provided however that the documentation is disseminated together with the software on the same medium. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Oracle. Oracle and/or its affiliates reserve any and all rights to this documentation not expressly granted above.

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 Introduction

This document describes what you need to know when working on the MySQL code. To track or contribute to MySQL development, follow the instructions in Installing MySQL Using a Development Source Tree. If you are interested in MySQL internals, you should also join the MySQL Community Slack. Feel free to ask questions about the code and to send patches that you would like to contribute to the MySQL project!

Chapter 2 MySQL Threads

The MySQL server creates the following threads:

- Connection manager threads handle client connection requests on the network interfaces that the server listens to. On all platforms, one manager thread handles TCP/IP connection requests. On Unix, this manager thread also handles Unix socket file connection requests. On Windows, a manager thread handles shared-memory connection requests, and another handles named-pipe connection requests. The server does not create threads to handle interfaces that it does not listen to. For example, a Windows server that does not have support for named-pipe connections enabled does not create a thread to handle them.
- Connection manager threads associate each client connection with a thread dedicated to it that handles authentication and request processing for that connection. Manager threads create a new thread when necessary but try to avoid doing so by consulting the thread cache first to see whether it contains a thread that can be used for the connection. When a connection ends, its thread is returned to the thread cache if the cache is not full.

  For information about tuning the parameters that control thread resources, see Connection Interfaces.
- On a source replication server, connections from replica servers are handled like client connections: There is one thread per connected replica.
- On a replica server, an I/O thread is started to connect to the source server and read updates from it. An SQL thread is started to apply updates read from the source. These two threads run independently and can be started and stopped independently.
- A signal thread handles all signals. This thread also normally handles alarms and calls `process_alarm()` to force timeouts on connections that have been idle too long.
- If `InnoDB` is used, there will be additional read and write threads by default. The number of these are controlled by the `innodb_read_io_threads` and `innodb_write_io_threads` parameters. See InnoDB Startup Options and System Variables.
- If the server is started with the `--flush_time=val` option, a dedicated thread is created to flush all tables every *`val`* seconds.
- If the event scheduler is active, there is one thread for the scheduler, and a thread for each event currently running. See Event Scheduler Overview.

`mysqladmin processlist` only shows the connection, replication, and event threads.

Chapter 3 The MySQL Test Suite

The test system that is included in Unix source and binary distributions makes it possible for users and developers to perform regression tests on the MySQL code. These tests can be run on Unix.

You can also write your own test cases. For information, including system requirements, see The MySQL Test Framework in the MySQL Server Doxygen documentation, available at https://dev.mysql.com/doc/index-other.html.

The current set of test cases does not test everything in MySQL, but it should catch most obvious bugs in the SQL processing code, operating system or library issues, and is quite thorough in testing replication. Our goal is to have the tests cover 100% of the code. We welcome contributions to our test suite. You may especially want to contribute tests that examine the functionality critical to your system because this ensures that all future MySQL releases work well with your applications.

The test system consists of a test language interpreter (`mysqltest`), a Perl script to run all tests (`mysql-test-run.pl`), the actual test cases written in a special test language, and their expected results. To run the test suite on your system after a build, type `make test` from the source root directory, or change location to the `mysql-test` directory and type `./mysql-test-run.pl`. If you have installed a binary distribution, change location to the `mysql-test` directory under the installation root directory (for example, `/usr/local/mysql/mysql-test`), and run `./mysql-test-run.pl`. All tests should succeed. If any do not, feel free to try to find out why and report the problem if it indicates a bug in MySQL. See How to Report Bugs or Problems.

If one test fails, you should run `mysql-test-run.pl` with the `--force` option to check whether any other tests fail.

If you have a copy of `mysqld` running on the machine where you want to run the test suite, you do not have to stop it, as long as it is not using ports `9306` or `9307`. If either of those ports is taken, you should set the `MTR_BUILD_THREAD` environment variable to an appropriate value, and the test suite will use a different set of ports for source, replica, and NDB). For example:

```
shell> export MTR_BUILD_THREAD=31
shell> ./mysql-test-run.pl [options] [test_name]
```

In the `mysql-test` directory, you can run an individual test case with `./mysql-test-run.pl` *`test_name`*.

If you have a question about the test suite, or have a test case to contribute, join the MySQL Community Slack.

Chapter 4 The MySQL Plugin API

Table of Contents

MySQL supports a plugin API that enables creation of server components. Plugins can be loaded at server startup, or loaded and unloaded at runtime without restarting the server. The API is generic and does not specify what plugins can do. The components supported by this interface include, but are not limited to, storage engines, full-text parser plugins, and server extensions.

For example, full-text parser plugins can be used to replace or augment the built-in full-text parser. A plugin can parse text into words using rules that differ from those used by the built-in parser. This can be useful if you need to parse text with characteristics different from those expected by the built-in parser.

The plugin interface is more general than the older loadable function interface.

The plugin interface uses the `plugin` table in the `mysql` database to record information about plugins that have been installed permanently with the `INSTALL PLUGIN` statement. This table is created as part of the MySQL installation process. Plugins can also be installed for a single server invocation with the `--plugin-load` option. Plugins installed this way are not recorded in the `plugin` table. See Installing and Uninstalling Plugins.

MySQL supports an API for client plugins in addition to that for server plugins. This is used, for example, by authentication plugins where a server-side plugin and a client-side plugin cooperate to enable clients to connect to the server through a variety of authentication methods.

Additional Resources

The book *MySQL 5.1 Plugin Development* by Sergei Golubchik and Andrew Hutchings provides a wealth of detail about the plugin API. Despite the fact that the book's title refers to MySQL Server 5.1, most of the information in it applies to later versions as well.

4.1 Types of Plugins

The plugin API enables creation of plugins that implement several capabilities:

- Storage engines
- Full-text parsers
- Daemons
- `INFORMATION_SCHEMA` tables
- Semisynchronous replication
- Auditing
- Authentication
- Password validation and strength checking
- Protocol tracing
- Query rewriting
- Secure keyring storage and retrieval

The following sections provide an overview of these plugin types.

- Storage Engine Plugins
- Full-Text Parser Plugins
- Daemon Plugins
- INFORMATION_SCHEMA Plugins
- Semisynchronous Replication Plugins
- Audit Plugins
- Authentication Plugins
- Password-Validation Plugins
- Protocol Trace Plugins
- Query Rewrite Plugins
- Keyring Plugins

Storage Engine Plugins

The pluggable storage engine architecture used by MySQL Server enables storage engines to be written as plugins and loaded into and unloaded from a running server. For a description of this architecture, see Overview of MySQL Storage Engine Architecture.

For information on how to use the plugin API to write storage engines, see MySQL Internals: Writing a Custom Storage Engine.

Full-Text Parser Plugins

MySQL has a built-in parser that it uses by default for full-text operations (parsing text to be indexed, or parsing a query string to determine the terms to be used for a search). The built-in full-text parser is supported with `InnoDB` and `MyISAM` tables.

MySQL also has a character-based ngram full-text parser that supports Chinese, Japanese, and Korean (CJK), and a word-based MeCab parser plugin that supports Japanese, for use with `InnoDB` and `MyISAM` tables.

For full-text processing, “parsing” means extracting words (or “tokens”, in the case of an n-gram character-based parser) from text or a query string based on rules that define which character sequences make up a word and where word boundaries lie.

When parsing for indexing purposes, the parser passes each word to the server, which adds it to a full-text index. When parsing a query string, the parser passes each word to the server, which accumulates the words for use in a search.

The parsing properties of the built-in full-text parser are described in Full-Text Search Functions. These properties include rules for determining how to extract words from text. The parser is influenced by certain system variables that cause words shorter or longer to be excluded, and by the stopword list that identifies common words to be ignored. For more information, see Full-Text Stopwords, and Fine-Tuning MySQL Full-Text Search.

The plugin API enables you to use a full-text parser other than the default built-in full-text parser. For example, if you are working with Japanese, you may choose to use the MeCab full-text parser. The plugin API also enables you to provide a full-text parser of your own so that you have control over the basic duties of a parser. A parser plugin can operate in either of two roles:

- The plugin can replace the built-in parser. In this role, the plugin reads the input to be parsed, splits it up into words, and passes the words to the server (either for indexing or for token accumulation). The ngram and MeCab parsers operate as replacements for the built-in full-text parser.

  You may choose to provide your own full-text parser if you need to use different rules from those of the built-in parser for determining how to split up input into words. For example, the built-in parser considers the text “case-sensitive” to consist of two words “case” and “sensitive,” whereas an application might need to treat the text as a single word.

- The plugin can act in conjunction with the built-in parser by serving as a front end for it. In this role, the plugin extracts text from the input and passes the text to the parser, which splits up the text into words using its normal parsing rules. This parsing is affected by the `innodb_ft_xxx` or `ft_xxx` system variables and the stopword list.

  One reason to use a parser this way is that you need to index content such as PDF documents, XML documents, or `.doc` files. The built-in parser is not intended for those types of input but a plugin can pull out the text from these input sources and pass it to the built-in parser.

It is also possible for a parser plugin to operate in both roles. That is, it could extract text from noncleartext input (the front end role), and also parse the text into words (thus replacing the built-in parser).

A full-text plugin is associated with full-text indexes on a per-index basis. That is, when you install a parser plugin initially, that does not cause it to be used for any full-text operations. It simply becomes available. For example, a full-text parser plugin becomes available to be named in a `WITH PARSER` clause when creating individual `FULLTEXT` indexes. To create such an index at table-creation time, do this:

```
CREATE TABLE t
(
  doc CHAR(255),
  FULLTEXT INDEX (doc) WITH PARSER parser_name
) ENGINE=InnoDB;
```

Or you can add the index after the table has been created:

```
ALTER TABLE t ADD FULLTEXT INDEX (doc) WITH PARSER parser_name;
```

The only SQL change for associating the parser with the index is the `WITH PARSER` clause. Searches are specified as before, with no changes needed for queries.

When you associate a parser plugin with a `FULLTEXT` index, the plugin is required for using the index. If the parser plugin is dropped, any index associated with it becomes unusable. Any attempt to use a table for which a plugin is not available results in an error, although `DROP TABLE` is still possible.

For more information about full-text plugins, see Section 4.4.4, “Writing Full-Text Parser Plugins”. MySQL 5.7 supports full-text plugins with `MyISAM` and `InnoDB`.

Daemon Plugins

A daemon plugin is a simple type of plugin used for code that should be run by the server but that does not communicate with it. MySQL distributions include an example daemon plugin that writes periodic heartbeat messages to a file.

For more information about daemon plugins, see Section 4.4.5, “Writing Daemon Plugins”.

INFORMATION_SCHEMA Plugins

`INFORMATION_SCHEMA` plugins enable the creation of tables containing server metadata that are exposed to users through the `INFORMATION_SCHEMA` database. For example, `InnoDB` uses `INFORMATION_SCHEMA` plugins to provide tables that contain information about current transactions and locks.

For more information about `INFORMATION_SCHEMA` plugins, see Section 4.4.6, “Writing INFORMATION_SCHEMA Plugins”.

Semisynchronous Replication Plugins

MySQL replication is asynchronous by default. With semisynchronous replication, a commit performed on the source side blocks before returning to the session that performed the transaction until at least one replica acknowledges that it has received and logged the events for the transaction. Semisynchronous replication is implemented through complementary source and client plugins. See Semisynchronous Replication.

For more information about semisynchronous replication plugins, see Section 4.4.7, “Writing Semisynchronous Replication Plugins”.

Audit Plugins

The MySQL server provides a pluggable audit interface that enables information about server operations to be reported to interested parties. Audit notification occurs for these operations (although the interface is general and the server could be modified to report others):

- Write a message to the general query log (if the log is enabled)
- Write a message to the error log
- Send a query result to a client

Audit plugins may register with the audit interface to receive notification about server operations. When an auditable event occurs within the server, the server determines whether notification is needed. For each registered audit plugin, the server checks the event against those event classes in which the plugin is interested and passes the event to the plugin if there is a match.

This interface enables audit plugins to receive notifications only about operations in event classes they consider significant and to ignore others. The interface provides for categorization of operations into event classes and further division into event subclasses within each class.

When an audit plugin is notified of an auditable event, it receives a pointer to the current THD structure and a pointer to a structure that contains information about the event. The plugin can examine the event and perform whatever auditing actions are appropriate. For example, the plugin can see what statement produced a result set or was logged, the number of rows in a result, who the current user was for an operation, or the error code for failed operations.

For more information about audit plugins, see Section 4.4.8, “Writing Audit Plugins”.

Authentication Plugins

MySQL supports pluggable authentication. Authentication plugins exist on both the server and client sides. Plugins on the server side implement authentication methods for use by clients when they connect to the server. A plugin on the client side communicates with a server-side plugin to provide the authentication information that it requires. A client-side plugin may interact with the user, performing tasks such as soliciting a password or other authentication credentials to be sent to the server. See Pluggable Authentication.

Pluggable authentication also enables proxy user capability, in which one user takes the identity of another user. A server-side authentication plugin can return to the server the name of the user whose identity the connecting user should have. See Proxy Users.

For more information about authentication plugins, see Section 4.4.9, “Writing Authentication Plugins”.

Password-Validation Plugins

The MySQL server provides an interface for writing plugins that test passwords. Such a plugin implements two capabilities:

- Rejection of too-weak passwords in statements that assign passwords (such as `CREATE USER` and `ALTER USER` statements), and passwords given as arguments to the `PASSWORD()` function.
- Assessing the strength of potential passwords for the `VALIDATE_PASSWORD_STRENGTH()` SQL function.

For information about writing this type of plugin, see Section 4.4.10, “Writing Password-Validation Plugins”.

Protocol Trace Plugins

MySQL supports the use of protocol trace plugins: client-side plugins that implement tracing of communication between a client and the server that takes place using the client/server protocol.

For more information about protocol trace plugins, see Section 4.4.11, “Writing Protocol Trace Plugins”.

Query Rewrite Plugins

MySQL Server supports query rewrite plugins that can examine and possibly modify statements received by the server before the server executes them. A query rewrite plugin takes statements either before or after the server has parsed them.

A preparse query rewrite plugin has these characteristics:

- The plugin enables rewriting of SQL statements arriving at the server before the server processes them.

- The plugin receives a statement string and may return a different string.

A postparse query rewrite plugin has these characteristics:

- The plugin enables statement rewriting based on parse trees.
- The server parses each statement and passes its parse tree to the plugin, which may traverse the tree. The plugin can return the original tree to the server for further processing, or construct a different tree and return that instead.
- The plugin can use the `mysql_parser` plugin service for these purposes:
  - To activate statement digest calculation and obtain the normalized version of statements independent of whether the Performance Schema produces digests.
  - To traverse parse trees.
  - To parse statements. This is useful if the plugin constructs a new statement string from the parse tree. The plugin can have the server parse the string to produce a new tree, then return that tree as the representation of the rewritten statement.

For more information about plugin services, see MySQL Plugin Services.

Preparse and postparse query rewrite plugins share these characteristics:

- If a query rewrite plugin is installed, the `--log-raw` option affects statement logging as follows:
  - Without `--log-raw`, the server logs the statement returned by the query rewrite plugin. This may differ from the statement as received.
  - With `--log-raw`, the server logs the original statement as received.
- If a plugin rewrites a statement, the server decides whether to write it to the binary log (and thus to any replicas) based on the rewritten statement, not the original statement. If a plugin rewrites only `SELECT` statements to `SELECT` statements, there is no impact on binary logging because the server does not write `SELECT` statements to the binary log.
- If a plugin rewrites a statement, the server produces a `Note` message that the client can view using `SHOW WARNINGS`. Messages have this format, where *`stmt_in`* is the original statement and *`stmt_out`* is the rewritten statement:

```
Query 'stmt_in' rewritten to 'stmt_out' by a query rewrite plugin
```

MySQL distributions include a postparse query rewrite plugin named `Rewriter`. This plugin is rule based. You can add rows to its rules table to cause `SELECT` statement rewriting. For more information, see The Rewriter Query Rewrite Plugin.

Query rewrite plugins use the same API as audit plugins. For more information about audit plugins, see Section 4.4.8, “Writing Audit Plugins”.

Keyring Plugins

As of MySQL 5.7.11, MySQL Server supports keyring plugins that enable internal server components and plugins to securely store sensitive information for later retrieval.

All MySQL distributions include a keyring plugin named `keyring_file`. MySQL Enterprise Edition distributions include additional keyring plugins. See The MySQL Keyring.

For more information about keyring plugins, see Section 4.4.12, “Writing Keyring Plugins”.

4.2 Plugin API Characteristics

The server plugin API has these characteristics:

- All plugins have several things in common.

  Each plugin has a name that it can be referred to in SQL statements, as well as other metadata such as an author and a description that provide other information. This information can be examined in the `INFORMATION_SCHEMA.PLUGINS` table or using the `SHOW PLUGINS` statement.

- The plugin framework is extendable to accommodate different kinds of plugins.

  Although some aspects of the plugin API are common to all types of plugins, the API also permits type-specific interface elements so that different types of plugins can be created. A plugin with one purpose can have an interface most appropriate to its own requirements and not the requirements of some other plugin type.

  Interfaces for several types of plugins exist, such as storage engines, full-text parser, and `INFORMATION_SCHEMA` tables. Others can be added.

- Plugins can expose information to users.

  A plugin can implement system and status variables that are available through the `SHOW VARIABLES` and `SHOW STATUS` statements.

- The plugin API includes versioning information.

  The version information included in the plugin API enables a plugin library and each plugin that it contains to be self-identifying with respect to the API version that was used to build the library. If the API changes over time, the version numbers will change, but a server can examine a given plugin library's version information to determine whether it supports the plugins in the library.

  There are two types of version numbers. The first is the version for the general plugin framework itself. Each plugin library includes this kind of version number. The second type of version applies to individual plugins. Each specific type of plugin has a version for its interface, so each plugin in a library has a type-specific version number. For example, a library containing a full-text parser plugin has a general plugin API version number, and the plugin has a version number specific to the full-text plugin interface.

- The plugin API implements security restrictions.

  A plugin library must be installed in a specific dedicated directory for which the location is controlled by the server and cannot be changed at runtime. Also, the library must contain specific symbols that identify it as a plugin library. The server will not load something as a plugin if it was not built as a plugin.

- Plugins have access to server services.

  The services interface exposes server functionality that plugins can access using ordinary function calls. For details, see MySQL Plugin Services.

In some respects, the server plugin API is similar to the older loadable function API that it supersedes, but the plugin API has several advantages over the older interface. For example, loadable functions had no versioning information. Also, the newer plugin interface eliminates the security issues of the older loadable function interface. The older interface for writing nonplugin loadable functions permitted libraries to be loaded from any directory searched by the system's dynamic linker, and the symbols that identified the loadable function library were relatively nonspecific.

The client plugin API has similar architectural characteristics, but client plugins have no direct access to the server the way server plugins do.

4.3 Plugin API Components

The server plugin implementation comprises several components.

SQL statements:

- `INSTALL PLUGIN` registers a plugin in the `mysql.plugin` table and loads the plugin code.
- `UNINSTALL PLUGIN` unregisters a plugin from the `mysql.plugin` table and unloads the plugin code.
- The `WITH PARSER` clause for full-text index creation associates a full-text parser plugin with a given `FULLTEXT` index.
- `SHOW PLUGINS` displays information about server plugins.

Command-line options and system variables:

- The `--plugin-load` option enables plugins to be loaded at server startup time.
- The `plugin_dir` system variable indicates the location of the directory where all plugins must be installed. The value of this variable can be specified at server startup with a `--plugin_dir=`*`dir_name`* option. `mysql_config --plugindir` displays the default plugin directory path name.

For additional information about plugin loading, see Installing and Uninstalling Plugins.

Plugin-related tables:

- The `INFORMATION_SCHEMA.PLUGINS` table contains plugin information.
- The `mysql.plugin` table lists each plugin that was installed with `INSTALL PLUGIN` and is required for plugin use. For new MySQL installations, this table is created during the installation process.

The client plugin implementation is simpler:

- For the `mysql_options()` C API function, the `MYSQL_DEFAULT_AUTH` and `MYSQL_PLUGIN_DIR` options enable client programs to load authentication plugins.
- There are C API functions that enable management of client plugins.

To examine how MySQL implements plugins, consult the following source files in a MySQL source distribution:

- In the `include/mysql` directory, `plugin.h` exposes the public plugin API. This file should be examined by anyone who wants to write a plugin library. `plugin_`*`xxx`*`.h` files provide additional information that pertains to specific types of plugins. `client_plugin.h` contains information specific to client plugins.
- In the `sql` directory, `sql_plugin.h` and `sql_plugin.cc` comprise the internal plugin implementation. `sql_acl.cc` is where the server uses authentication plugins. These files need not be consulted by plugin developers. They may be of interest for those who want to know more about how the server handles plugins.
- In the `sql-common` directory, `client_plugin.h` implements the C API client plugin functions, and `client.c` implements client authentication support. These files need not be consulted by plugin

developers. They may be of interest for those who want to know more about how the server handles plugins.

4.4 Writing Plugins

To create a plugin library, you must provide the required descriptor information that indicates what plugins the library file contains, and write the interface functions for each plugin.

Every server plugin must have a general descriptor that provides information to the plugin API, and a type-specific descriptor that provides information about the plugin interface for a given type of plugin. The structure of the general descriptor is the same for all plugin types. The structure of the type-specific descriptor varies among plugin types and is determined by the requirements of what the plugin needs to do. The server plugin interface also enables plugins to expose status and system variables. These variables become visible through the `SHOW STATUS` and `SHOW VARIABLES` statements and the corresponding `INFORMATION_SCHEMA` tables.

For client-side plugins, the architecture is a bit different. Each plugin must have a descriptor, but there is no division into separate general and type-specific descriptors. Instead, the descriptor begins with a fixed set of members common to all client plugin types, and the common members are followed by any additional members required to implement the specific plugin type.

You can write plugins in C or C++ (or another language that can use C calling conventions). Plugins are loaded and unloaded dynamically, so your operating system must support dynamic loading and you must have compiled the calling application dynamically (not statically). For server plugins, this means that `mysqld` must be linked dynamically.

A server plugin contains code that becomes part of the running server, so when you write the plugin, you are bound by any and all constraints that otherwise apply to writing server code. For example, you may have problems if you attempt to use functions from the `libstdc++` library. These constraints may change in future versions of the server, so it is possible that server upgrades will require revisions to plugins originally written for older servers. For information about these constraints, see MySQL Source-Configuration Options, and Dealing with Problems Compiling MySQL.

Client plugin writers should avoid dependencies on what symbols the calling application has because you cannot be sure what applications will use the plugin.

4.4.1 Overview of Plugin Writing

The following procedure provides an overview of the steps needed to create a plugin library. The next sections provide additional details on setting plugin data structures and writing specific types of plugins.

1. In the plugin source file, include the header files that the plugin library needs. The `plugin.h` file is required, and the library might require other files as well. For example:

   ```
   #include <stdlib.h>
   #include <ctype.h>
   #include <mysql/plugin.h>
   ```

2. Set up the descriptor information for the plugin library file. For server plugins, write the library descriptor, which must contain the general plugin descriptor for each server plugin in the file. For more information, see Section 4.4.2.1, “Server Plugin Library and Plugin Descriptors”. In addition, set up the type-specific descriptor for each server plugin in the library. Each plugin's general descriptor points to its type-specific descriptor.

   For client plugins, write the client descriptor. For more information, see Section 4.4.2.3, “Client Plugin Descriptors”.

3. Write the plugin interface functions for each plugin. For example, each plugin's general plugin descriptor points to the initialization and deinitialization functions that the server should invoke when it loads and unloads the plugin. The plugin's type-specific description may also point to interface functions.

4. For server plugins, set up the status and system variables, if there are any.

5. Compile the plugin library as a shared library and install it in the plugin directory. For more information, see Section 4.4.3, “Compiling and Installing Plugin Libraries”.

6. For server plugins, register the plugin with the server. For more information, see Installing and Uninstalling Plugins.

7. Test the plugin to verify that it works properly.

4.4.2 Plugin Data Structures

A plugin library file includes descriptor information to indicate what plugins it contains.

If the plugin library contains any server plugins, it must include the following descriptor information:

- A library descriptor indicates the general server plugin API version number used by the library and contains a general plugin descriptor for each server plugin in the library. To provide the framework for this descriptor, invoke two macros from the `plugin.h` header file:

  ```
  mysql_declare_plugin(name)
   ... one or more server plugin descriptors here ...
  mysql_declare_plugin_end;
  ```

  The macros expand to provide a declaration for the API version automatically. You must provide the plugin descriptors.

- Within the library descriptor, each general server plugin is described by a `st_mysql_plugin` structure. This plugin descriptor structure contains information that is common to every type of server plugin: A value that indicates the plugin type; the plugin name, author, description, and license type; pointers to the initialization and deinitialization functions that the server invokes when it loads and unloads the plugin, and pointers to any status or system variables the plugin implements.

- Each general server plugin descriptor within the library descriptor also contains a pointer to a type-specific plugin descriptor. The structure of the type-specific descriptors varies from one plugin type to another because each type of plugin can have its own API. A type-specific plugin descriptor contains a type-specific API version number and pointers to the functions that are needed to implement that plugin type. For example, a full-text parser plugin has initialization and deinitialization functions, and a main parsing function. The server invokes these functions when it uses the plugin to parse text.

The plugin library also contains the interface functions that are referenced by the general and type-specific descriptors for each plugin in the library.

If the plugin library contains a client plugin, it must include a descriptor for the plugin. The descriptor begins with a fixed set of members common to all client plugins, followed by any members specific to the plugin type. To provide the descriptor framework, invoke two macros from the `client_plugin.h` header file:

```
mysql_declare_client_plugin(plugin_type)
   ... members common to all client plugins ...
   ... type-specific extra members ...
mysql_end_client_plugin;
```

The plugin library also contains any interface functions referenced by the client descriptor.

The `mysql_declare_plugin()` and `mysql_declare_client_plugin()` macros differ somewhat in how they can be invoked, which has implications for the contents of plugin libraries. The following guidelines summarize the rules:

- `mysql_declare_plugin()` and `mysql_declare_client_plugin()` can both be used in the same source file, which means that a plugin library can contain both server and client plugins. However, each of `mysql_declare_plugin()` and `mysql_declare_client_plugin()` can be used at most once.
- `mysql_declare_plugin()` permits multiple server plugin declarations, so a plugin library can contain multiple server plugins.
- `mysql_declare_client_plugin()` permits only a single client plugin declaration. To create multiple client plugins, separate plugin libraries must be used.

When a client program looks for a client plugin that is in a plugin library and not built into `libmysqlclient`, it looks for a file with a base name that is the same as the plugin name. For example, if a program needs to use a client authentication plugin named `auth_xxx` on a system that uses `.so` as the library suffix, it looks in the file named `auth_xxx.so`. (On macOS, the program looks first for `auth_xxx.dylib`, then for `auth_xxx.so`.) For this reason, if a plugin library contains a client plugin, the library must have the same base name as that plugin.

The same is not true for a library that contains server plugins. The `--plugin-load` option and the `INSTALL PLUGIN` statement provide the library file name explicitly, so there need be no explicit relationship between the library name and the name of any server plugins it contains.

4.4.2.1 Server Plugin Library and Plugin Descriptors

Every plugin library that contains server plugins must include a library descriptor that contains the general plugin descriptor for each server plugin in the file. This section discusses how to write the library and general descriptors for server plugins.

The library descriptor must define two symbols:

- `_mysql_plugin_interface_version_` specifies the version number of the general plugin framework. This is given by the `MYSQL_PLUGIN_INTERFACE_VERSION` symbol, which is defined in the `plugin.h` file.
- `_mysql_plugin_declarations_` defines an array of plugin declarations, terminated by a declaration with all members set to 0. Each declaration is an instance of the `st_mysql_plugin` structure (also defined in `plugin.h`). There must be one of these for each server plugin in the library.

If the server does not find those two symbols in a library, it does not accept it as a legal plugin library and rejects it with an error. This prevents use of a library for plugin purposes unless it was built specifically as a plugin library.

The conventional way to define the two required symbols is by using the `mysql_declare_plugin()` and `mysql_declare_plugin_end` macros from the `plugin.h` file:

```
mysql_declare_plugin(name)
 ... one or more server plugin descriptors here ...
mysql_declare_plugin_end;
```

Each server plugin must have a general descriptor that provides information to the server plugin API. The general descriptor has the same structure for all plugin types. The `st_mysql_plugin` structure in the `plugin.h` file defines this descriptor:

```
struct st_mysql_plugin
{
  int type;             /* the plugin type (a MYSQL_XXX_PLUGIN value)   */
```

```
  void *info;           /* pointer to type-specific plugin descriptor   */
  const char *name;     /* plugin name                                  */
  const char *author;   /* plugin author (for I_S.PLUGINS)              */
  const char *descr;    /* general descriptive text (for I_S.PLUGINS)   */
  int license;          /* the plugin license (PLUGIN_LICENSE_XXX)      */
  int (*init)(void *);  /* the function to invoke when plugin is loaded */
  int (*deinit)(void *);/* the function to invoke when plugin is unloaded */
  unsigned int version; /* plugin version (for I_S.PLUGINS)             */
  struct st_mysql_show_var *status_vars;
  struct st_mysql_sys_var **system_vars;
  void * __reserved1;   /* reserved for dependency checking             */
  unsigned long flags;  /* flags for plugin */
};
```

The `st_mysql_plugin` descriptor structure members are used as follows. `char *` members should be specified as null-terminated strings.

- `type`: The plugin type. This must be one of the plugin-type values from `plugin.h`:

  ```
  /*
    The allowable types of plugins
  */
  #define MYSQL_UDF_PLUGIN              0  /* User-defined function        */
  #define MYSQL_STORAGE_ENGINE_PLUGIN   1  /* Storage Engine               */
  #define MYSQL_FTPARSER_PLUGIN         2  /* Full-text parser plugin      */
  #define MYSQL_DAEMON_PLUGIN           3  /* The daemon/raw plugin type */
  #define MYSQL_INFORMATION_SCHEMA_PLUGIN  4  /* The I_S plugin type */
  #define MYSQL_AUDIT_PLUGIN            5  /* The Audit plugin type        */
  #define MYSQL_REPLICATION_PLUGIN      6  /* The replication plugin type */
  #define MYSQL_AUTHENTICATION_PLUGIN   7  /* The authentication plugin type */
  ...
  ```

  For example, for a full-text parser plugin, the `type` value is `MYSQL_FTPARSER_PLUGIN`.

- `info`: A pointer to the type-specific descriptor for the plugin. This descriptor's structure depends on the particular type of plugin, unlike that of the general plugin descriptor structure. For version-control purposes, the first member of the type-specific descriptor for every plugin type is expected to be the interface version for the type. This enables the server to check the type-specific version for every plugin no matter its type. Following the version number, the descriptor includes any other members needed, such as callback functions and other information needed by the server to invoke the plugin properly. Later sections on writing particular types of server plugins describe the structure of their type-specific descriptors.

- `name`: A string that gives the plugin name. This is the name that will be listed in the `mysql.plugin` table and by which you refer to the plugin in SQL statements such as `INSTALL PLUGIN` and `UNINSTALL PLUGIN`, or with the `--plugin-load` option. The name is also visible in the `INFORMATION_SCHEMA.PLUGINS` table or the output from `SHOW PLUGINS`.

  The plugin name should not begin with the name of any server option. If it does, the server will fail to initialize it. For example, the server has a `--socket` option, so you should not use a plugin name such as `socket`, `socket_plugin`, and so forth.

- `author`: A string naming the plugin author. This can be whatever you like.

- `desc`: A string that provides a general description of the plugin. This can be whatever you like.

- `license`: The plugin license type. The value can be one of `PLUGIN_LICENSE_PROPRIETARY`, `PLUGIN_LICENSE_GPL`, or `PLUGIN_LICENSE_BSD`.

- `init`: A once-only initialization function, or `NULL` if there is no such function. The server executes this function when it loads the plugin, which happens for `INSTALL PLUGIN` or, for plugins listed in the

`mysql.plugin` table, at server startup. The function takes one argument that points to the internal structure used to identify the plugin. It returns zero for success and nonzero for failure.

- `deinit`: A once-only deinitialization function, or `NULL` if there is no such function. The server executes this function when it unloads the plugin, which happens for `UNINSTALL PLUGIN` or, for plugins listed in the `mysql.plugin` table, at server shutdown. The function takes one argument that points to the internal structure used to identify the plugin It returns zero for success and nonzero for failure.

- `version`: The plugin version number. When the plugin is installed, this value can be retrieved from the `INFORMATION_SCHEMA.PLUGINS` table. The value includes major and minor numbers. If you write the value as a hex constant, the format is `0x`*`MMNN`*, where *`MM`* and `NN` are the major and minor numbers, respectively. For example, `0x0302` represents version 3.2.

- `status_vars`: A pointer to a structure for status variables associated with the plugin, or `NULL` if there are no such variables. When the plugin is installed, these variables are displayed in the output of the `SHOW STATUS` statement.

  The `status_vars` member, if not `NULL`, points to an array of `st_mysql_show_var` structures that describe status variables. See Section 4.4.2.2, “Server Plugin Status and System Variables”.

- `system_vars`: A pointer to a structure for system variables associated with the plugin, or `NULL` if there are no such variables. These options and system variables can be used to help initialize variables within the plugin. When the plugin is installed, these variables are displayed in the output of the `SHOW VARIABLES` statement.

  The `system_vars` member, if not `NULL`, points to an array of `st_mysql_sys_var` structures that describe system variables. See Section 4.4.2.2, “Server Plugin Status and System Variables”.

- `__reserved1`: A placeholder for the future. It should be set to `NULL`.

- `flags`: Plugin flags. Individual bits correspond to different flags. The value should be set to the OR of the applicable flags. These flags are available:

  ```
  #define PLUGIN_OPT_NO_INSTALL   1UL   /* Not dynamically loadable */
  #define PLUGIN_OPT_NO_UNINSTALL 2UL   /* Not dynamically unloadable */
  ```

  The flags have the following meanings when enabled:

  - `PLUGIN_OPT_NO_INSTALL`: The plugin cannot be loaded at runtime with the `INSTALL PLUGIN` statement. This is appropriate for plugins that must be loaded at server startup with the `--plugin-load` or `--plugin-load-add` option.

  - `PLUGIN_OPT_NO_UNINSTALL`: The plugin cannot be unloaded at runtime with the `UNINSTALL PLUGIN` statement.

The server invokes the `init` and `deinit` functions in the general plugin descriptor only when loading and unloading the plugin. They have nothing to do with use of the plugin such as happens when an SQL statement causes the plugin to be invoked.

For example, the descriptor information for a library that contains a single full-text parser plugin named `simple_parser` looks like this:

```
mysql_declare_plugin(ftexample)
{
  MYSQL_FTPARSER_PLUGIN,      /* type                            */
  &simple_parser_descriptor,  /* descriptor                      */
  "simple_parser",            /* name                            */
  "Oracle Corporation",       /* author                          */
```

```
  "Simple Full-Text Parser",  /* description                    */
  PLUGIN_LICENSE_GPL,         /* plugin license                 */
  simple_parser_plugin_init,  /* init function (when loaded)    */
  simple_parser_plugin_deinit,/* deinit function (when unloaded) */
  0x0001,                     /* version                        */
  simple_status,              /* status variables               */
  simple_system_variables,    /* system variables               */
  NULL,
  0
}
mysql_declare_plugin_end;
```

For a full-text parser plugin, the type must be `MYSQL_FTPARSER_PLUGIN`. This is the value that identifies the plugin as being legal for use in a `WITH PARSER` clause when creating a `FULLTEXT` index. (No other plugin type is legal for this clause.)

`plugin.h` defines the `mysql_declare_plugin()` and `mysql_declare_plugin_end` macros like this:

```
#ifndef MYSQL_DYNAMIC_PLUGIN
#define __MYSQL_DECLARE_PLUGIN(NAME, VERSION, PSIZE, DECLS) \
MYSQL_PLUGIN_EXPORT int VERSION= MYSQL_PLUGIN_INTERFACE_VERSION; \
MYSQL_PLUGIN_EXPORT int PSIZE= sizeof(struct st_mysql_plugin); \
MYSQL_PLUGIN_EXPORT struct st_mysql_plugin DECLS[]= {
#else
#define __MYSQL_DECLARE_PLUGIN(NAME, VERSION, PSIZE, DECLS) \
MYSQL_PLUGIN_EXPORT int _mysql_plugin_interface_version_= MYSQL_PLUGIN_INTERFACE_VERSION; \
MYSQL_PLUGIN_EXPORT int _mysql_sizeof_struct_st_plugin_= sizeof(struct st_mysql_plugin); \
MYSQL_PLUGIN_EXPORT struct st_mysql_plugin _mysql_plugin_declarations_[]= {
#endif

#define mysql_declare_plugin(NAME) \
__MYSQL_DECLARE_PLUGIN(NAME, \
                builtin_ ## NAME ## _plugin_interface_version, \
                builtin_ ## NAME ## _sizeof_struct_st_plugin, \
                builtin_ ## NAME ## _plugin)

#define mysql_declare_plugin_end ,{0,0,0,0,0,0,0,0,0,0,0,0,0}}
```

> **Note**
>
> Those declarations define the `_mysql_plugin_interface_version_` symbol only if the `MYSQL_DYNAMIC_PLUGIN` symbol is defined. This means that `-DMYSQL_DYNAMIC_PLUGIN` must be provided as part of the compilation command to build the plugin as a shared library.

When the macros are used as just shown, they expand to the following code, which defines both of the required symbols (`_mysql_plugin_interface_version_` and `_mysql_plugin_declarations_`):

```
int _mysql_plugin_interface_version_= MYSQL_PLUGIN_INTERFACE_VERSION;
int _mysql_sizeof_struct_st_plugin_= sizeof(struct st_mysql_plugin);
struct st_mysql_plugin _mysql_plugin_declarations_[]= {
{
  MYSQL_FTPARSER_PLUGIN,      /* type                           */
  &simple_parser_descriptor,  /* descriptor                     */
  "simple_parser",            /* name                           */
  "Oracle Corporation",       /* author                         */
  "Simple Full-Text Parser",  /* description                    */
  PLUGIN_LICENSE_GPL,         /* plugin license                 */
  simple_parser_plugin_init,  /* init function (when loaded)    */
  simple_parser_plugin_deinit,/* deinit function (when unloaded) */
  0x0001,                     /* version                        */
  simple_status,              /* status variables               */
  simple_system_variables,    /* system variables               */
```

```
  NULL,
  0
}
  ,{0,0,0,0,0,0,0,0,0,0,0,0}}
};
```

The preceding example declares a single plugin in the general descriptor, but it is possible to declare multiple plugins. List the declarations one after the other between `mysql_declare_plugin()` and `mysql_declare_plugin_end`, separated by commas.

MySQL server plugins can be written in C or C++ (or another language that can use C calling conventions). If you write a C++ plugin, one C++ feature that you should not use is nonconstant variables to initialize global structures. Members of structures such as the `st_mysql_plugin` structure should be initialized only with constant variables. The `simple_parser` descriptor shown earlier is permissible in a C++ plugin because it satisfies that requirement:

```
mysql_declare_plugin(ftexample)
{
  MYSQL_FTPARSER_PLUGIN,      /* type                            */
  &simple_parser_descriptor,  /* descriptor                      */
  "simple_parser",            /* name                            */
  "Oracle Corporation",       /* author                          */
  "Simple Full-Text Parser",  /* description                     */
  PLUGIN_LICENSE_GPL,         /* plugin license                  */
  simple_parser_plugin_init,  /* init function (when loaded)     */
  simple_parser_plugin_deinit,/* deinit function (when unloaded) */
  0x0001,                     /* version                         */
  simple_status,              /* status variables                */
  simple_system_variables,    /* system variables                */
  NULL,
  0
}
mysql_declare_plugin_end;
```

Here is another valid way to write the general descriptor. It uses constant variables to indicate the plugin name, author, and description:

```
const char *simple_parser_name = "simple_parser";
const char *simple_parser_author = "Oracle Corporation";
const char *simple_parser_description = "Simple Full-Text Parser";

mysql_declare_plugin(ftexample)
{
  MYSQL_FTPARSER_PLUGIN,      /* type                            */
  &simple_parser_descriptor,  /* descriptor                      */
  simple_parser_name,         /* name                            */
  simple_parser_author,       /* author                          */
  simple_parser_description,  /* description                     */
  PLUGIN_LICENSE_GPL,         /* plugin license                  */
  simple_parser_plugin_init,  /* init function (when loaded)     */
  simple_parser_plugin_deinit,/* deinit function (when unloaded) */
  0x0001,                     /* version                         */
  simple_status,              /* status variables                */
  simple_system_variables,    /* system variables                */
  NULL,
  0
}
mysql_declare_plugin_end;
```

However, the following general descriptor is invalid. It uses structure members to indicate the plugin name, author, and description, but structures are not considered constant initializers in C++:

```
typedef struct
{
```

```
  const char *name;
  const char *author;
  const char *description;
} plugin_info;

plugin_info parser_info = {
  "simple_parser",
  "Oracle Corporation",
  "Simple Full-Text Parser"
};

mysql_declare_plugin(ftexample)
{
  MYSQL_FTPARSER_PLUGIN,      /* type                               */
  &simple_parser_descriptor,  /* descriptor                         */
  parser_info.name,           /* name                               */
  parser_info.author,         /* author                             */
  parser_info.description,    /* description                        */
  PLUGIN_LICENSE_GPL,         /* plugin license                     */
  simple_parser_plugin_init,  /* init function (when loaded)     */
  simple_parser_plugin_deinit,/* deinit function (when unloaded) */
  0x0001,                     /* version                            */
  simple_status,              /* status variables                   */
  simple_system_variables,    /* system variables                   */
  NULL,
  0
}
mysql_declare_plugin_end;
```

4.4.2.2 Server Plugin Status and System Variables

The server plugin interface enables plugins to expose status and system variables using the `status_vars` and `system_vars` members of the general plugin descriptor.

The `status_vars` member of the general plugin descriptor, if not 0, points to an array of `st_mysql_show_var` structures, each of which describes one status variable, followed by a structure with all members set to 0. The `st_mysql_show_var` structure has this definition:

```
struct st_mysql_show_var {
  const char *name;
  char *value;
  enum enum_mysql_show_type type;
};
```

The following table shows the permissible status variable `type` values and what the corresponding variable should be.

Table 4.1 Server Plugin Status Variable Types

Variable Type	Meaning
`SHOW_BOOL`	Pointer to a boolean variable
`SHOW_INT`	Pointer to an integer variable
`SHOW_LONG`	Pointer to a long integer variable
`SHOW_LONGLONG`	Pointer to a longlong integer variable
`SHOW_CHAR`	A string
`SHOW_CHAR_PTR`	Pointer to a string
`SHOW_ARRAY`	Pointer to another `st_mysql_show_var` array
`SHOW_FUNC`	Pointer to a function

Variable Type	Meaning
SHOW_DOUBLE	Pointer to a double

For the SHOW_FUNC type, the function is called and fills in its out parameter, which then provides information about the variable to be displayed. The function has this signature:

```
#define SHOW_VAR_FUNC_BUFF_SIZE 1024

typedef int (*mysql_show_var_func) (void *thd,
                                    struct st_mysql_show_var *out,
                                    char *buf);
```

The system_vars member, if not 0, points to an array of st_mysql_sys_var structures, each of which describes one system variable (which can also be set from the command-line or configuration file), followed by a structure with all members set to 0. The st_mysql_sys_var structure is defined as follows:

```
struct st_mysql_sys_var {
 int flags;
 const char *name, *comment;
 int (*check)(THD*, struct st_mysql_sys_var *, void*, st_mysql_value*);
 void (*update)(THD*, struct st_mysql_sys_var *, void*, const void*);
};
```

Additional fields are append as required depending upon the flags.

For convenience, a number of macros are defined that make creating new system variables within a plugin much simpler.

Throughout the macros, the following fields are available:

- name: An unquoted identifier for the system variable.
- varname: The identifier for the static variable. Where not available, it is the same as the name field.
- opt: Additional use flags for the system variable. The following table shows the permissible flags.

Table 4.2 Server Plugin System Variable Flags

Flag Value	Description
PLUGIN_VAR_READONLY	The system variable is read only
PLUGIN_VAR_NOSYSVAR	The system variable is not user visible at runtime
PLUGIN_VAR_NOCMDOPT	The system variable is not configurable from the command line
PLUGIN_VAR_NOCMDARG	No argument is required at the command line (typically used for boolean variables)
PLUGIN_VAR_RQCMDARG	An argument is required at the command line (this is the default)
PLUGIN_VAR_OPCMDARG	An argument is optional at the command line
PLUGIN_VAR_MEMALLOC	Used for string variables; indicates that memory is to be allocated for storage of the string

- comment: A descriptive comment to be displayed in the server help message. NULL if this variable is to be hidden.
- check: The check function, NULL for default.

- `update`: The update function, `NULL` for default.
- `default`: The variable default value.
- `minimum`: The variable minimum value.
- `maximum`: The variable maximum value.
- `blocksize`: The variable block size. When the value is set, it is rounded to the nearest multiple of `blocksize`.

A system variable may be accessed either by using the static variable directly or by using the `SYSVAR()`accessor macro. The `SYSVAR()` macro is provided for completeness. Usually it should be used only when the code cannot directly access the underlying variable.

For example:

```
static int my_foo;
static MYSQL_SYSVAR_INT(foo_var, my_foo,
                        PLUGIN_VAR_RQCMDARG, "foo comment",
                        NULL, NULL, 0, 0, INT_MAX, 0);
 ...
   SYSVAR(foo_var)= value;
   value= SYSVAR(foo_var);
   my_foo= value;
   value= my_foo;
```

Session variables may be accessed only through the `THDVAR()` accessor macro. For example:

```
static MYSQL_THDVAR_BOOL(some_flag,
                         PLUGIN_VAR_NOCMDARG, "flag comment",
                         NULL, NULL, FALSE);
 ...
   if (THDVAR(thd, some_flag))
   {
     do_something();
     THDVAR(thd, some_flag)= FALSE;
   }
```

All global and session system variables must be published to `mysqld` before use. This is done by constructing a `NULL`-terminated array of the variables and linking to it in the plugin public interface. For example:

```
static struct st_mysql_sys_var *my_plugin_vars[]= {
  MYSQL_SYSVAR(foo_var),
  MYSQL_SYSVAR(some_flag),
  NULL
};
mysql_declare_plugin(fooplug)
{
  MYSQL_..._PLUGIN,
  &plugin_data,
  "fooplug",
  "foo author",
  "This does foo!",
  PLUGIN_LICENSE_GPL,
  foo_init,
  foo_fini,
  0x0001,
  NULL,
  my_plugin_vars,
  NULL,
  0
```

```
}
mysql_declare_plugin_end;
```

The following convenience macros enable you to declare different types of system variables:

- Boolean system variables of type `my_bool`, which is a 1-byte boolean. (0 = FALSE, 1 = TRUE)

```
MYSQL_THDVAR_BOOL(name, opt, comment, check, update, default)
MYSQL_SYSVAR_BOOL(name, varname, opt, comment, check, update, default)
```

- String system variables of type `char*`, which is a pointer to a null-terminated string.

```
MYSQL_THDVAR_STR(name, opt, comment, check, update, default)
MYSQL_SYSVAR_STR(name, varname, opt, comment, check, update, default)
```

- Integer system variables, of which there are several varieties.
  - An `int` system variable, which is typically a 4-byte signed word.

```
MYSQL_THDVAR_INT(name, opt, comment, check, update, default, min, max, blk)
MYSQL_SYSVAR_INT(name, varname, opt, comment, check, update, default,
                 minimum, maximum, blocksize)
```

  - An `unsigned int` system variable, which is typically a 4-byte unsigned word.

```
MYSQL_THDVAR_UINT(name, opt, comment, check, update, default, min, max, blk)
MYSQL_SYSVAR_UINT(name, varname, opt, comment, check, update, default,
                  minimum, maximum, blocksize)
```

  - A `long` system variable, which is typically either a 4- or 8-byte signed word.

```
MYSQL_THDVAR_LONG(name, opt, comment, check, update, default, min, max, blk)
MYSQL_SYSVAR_LONG(name, varname, opt, comment, check, update, default,
                  minimum, maximum, blocksize)
```

  - An `unsigned long` system variable, which is typically either a 4- or 8-byte unsigned word.

```
MYSQL_THDVAR_ULONG(name, opt, comment, check, update, default, min, max, blk)
MYSQL_SYSVAR_ULONG(name, varname, opt, comment, check, update, default,
                   minimum, maximum, blocksize)
```

  - A `long long` system variable, which is typically an 8-byte signed word.

```
MYSQL_THDVAR_LONGLONG(name, opt, comment, check, update,
                      default, minimum, maximum, blocksize)
MYSQL_SYSVAR_LONGLONG(name, varname, opt, comment, check, update,
                      default, minimum, maximum, blocksize)
```

  - An `unsigned long long` system variable, which is typically an 8-byte unsigned word.

```
MYSQL_THDVAR_ULONGLONG(name, opt, comment, check, update,
                       default, minimum, maximum, blocksize)
MYSQL_SYSVAR_ULONGLONG(name, varname, opt, comment, check, update,
                       default, minimum, maximum, blocksize)
```

  - A `double` system variable, which is typically an 8-byte signed word.

```
MYSQL_THDVAR_DOUBLE(name, opt, comment, check, update,
                    default, minimum, maximum, blocksize)
MYSQL_SYSVAR_DOUBLE(name, varname, opt, comment, check, update,
                    default, minimum, maximum, blocksize)
```

  - An `unsigned long` system variable, which is typically either a 4- or 8-byte unsigned word. The range of possible values is an ordinal of the number of elements in the `typelib`, starting from 0.

```
MYSQL_THDVAR_ENUM(name, opt, comment, check, update, default, typelib)
MYSQL_SYSVAR_ENUM(name, varname, opt, comment, check, update,
                  default, typelib)
```

- An `unsigned long long` system variable, which is typically an 8-byte unsigned word. Each bit represents an element in the `typelib`.

```
MYSQL_THDVAR_SET(name, opt, comment, check, update, default, typelib)
MYSQL_SYSVAR_SET(name, varname, opt, comment, check, update,
                 default, typelib)
```

Internally, all mutable and plugin system variables are stored in a `HASH` structure.

Display of the server command-line help text is handled by compiling a `DYNAMIC_ARRAY` of all variables relevant to command-line options, sorting them, and then iterating through them to display each option.

When a command-line option has been handled, it is then removed from the `argv` by the `handle_option()` function (`my_getopt.c`); in effect, it is consumed.

The server processes command-line options during the plugin installation process, immediately after the plugin has been successfully loaded but before the plugin initialization function has been called

Plugins loaded at runtime do not benefit from any configuration options and must have usable defaults. Once they are installed, they are loaded at `mysqld` initialization time and configuration options can be set at the command line or within `my.cnf`.

Plugins should consider the `thd` parameter to be read only.

4.4.2.3 Client Plugin Descriptors

Each client plugin must have a descriptor that provides information to the client plugin API. The descriptor structure begins with a fixed set of members common to all client plugins, followed by any members specific to the plugin type.

The `st_mysql_client_plugin` structure in the `client_plugin.h` file defines a “generic” descriptor that contains the common members:

```
struct st_mysql_client_plugin
{
  int type;
  unsigned int interface_version;
  const char *name;
  const char *author;
  const char *desc;
  unsigned int version[3];
  const char *license;
  void *mysql_api;
  int (*init)(char *, size_t, int, va_list);
  int (*deinit)();
  int (*options)(const char *option, const void *);
};
```

The common `st_mysql_client_plugin` descriptor structure members are used as follows. `char *` members should be specified as null-terminated strings.

- `type`: The plugin type. This must be one of the plugin-type values from `client_plugin.h`, such as `MYSQL_CLIENT_AUTHENTICATION_PLUGIN`.
- `interface_version`: The plugin interface version. For example, this is `MYSQL_CLIENT_AUTHENTICATION_PLUGIN_INTERFACE_VERSION` for an authentication plugin.

- `name`: A string that gives the plugin name. This is the name by which you refer to the plugin when you call `mysql_options()` with the `MYSQL_DEFAULT_AUTH` option or specify the `--default-auth` option to a MySQL client program.
- `author`: A string naming the plugin author. This can be whatever you like.
- `desc`: A string that provides a general description of the plugin. This can be whatever you like.
- `version`: The plugin version as an array of three integers indicating the major, minor, and teeny versions. For example, `{1,2,3}` indicates version 1.2.3.
- `license`: A string that specifies the license type.
- `mysql_api`: For internal use. Specify it as `NULL` in the plugin descriptor.
- `init`: A once-only initialization function, or `NULL` if there is no such function. The client library executes this function when it loads the plugin. The function returns zero for success and nonzero for failure.

  The `init` function uses its first two arguments to return an error message if an error occurs. The first argument is a pointer to a `char` buffer, and the second argument indicates the buffer length. Any message returned by the `init` function must be null-terminated, so the maximum message length is the buffer length minus one. The next arguments are passed to `mysql_load_plugin()`. The first indicates how many more arguments there are (0 if none), followed by any remaining arguments.
- `deinit`: A once-only deinitialization function, or `NULL` if there is no such function. The client library executes this function when it unloads the plugin. The function takes no arguments. It returns zero for success and nonzero for failure.
- `options`: A function for handling options passed to the plugin, or `NULL` if there is no such function. The function takes two arguments representing the option name and a pointer to its value. The function returns zero for success and nonzero for failure.

For a given client plugin type, the common descriptor members may be followed by additional members necessary to implement plugins of that type. For example, the `st_mysql_client_plugin_AUTHENTICATION` structure for authentication plugins has a function at the end that the client library calls to perform authentication.

To declare a plugin, use the `mysql_declare_client_plugin()` and `mysql_end_client_plugin` macros:

```
mysql_declare_client_plugin(plugin_type)
   ... members common to all client plugins ...
   ... type-specific extra members ...
mysql_end_client_plugin;
```

Do not specify the `type` or `interface_version` member explicitly. The `mysql_declare_client_plugin()` macro uses the *`plugin_type`* argument to generate their values automatically. For example, declare an authentication client plugin like this:

```
mysql_declare_client_plugin(AUTHENTICATION)
  "my_auth_plugin",
  "Author Name",
  "My Client Authentication Plugin",
  {1,0,0},
  "GPL",
  NULL,
  my_auth_init,
  my_auth_deinit,
  my_auth_options,
```

```
  my_auth_main
mysql_end_client_plugin;
```

This declaration uses the `AUTHENTICATION` argument to set the `type` and `interface_version` members to `MYSQL_CLIENT_AUTHENTICATION_PLUGIN` and `MYSQL_CLIENT_AUTHENTICATION_PLUGIN_INTERFACE_VERSION`.

Depending on the plugin type, the descriptor may have other members following the common members. For example, for an authentication plugin, there is a function (`my_auth_main()` in the descriptor just shown) that handles communication with the server. See Section 4.4.9, “Writing Authentication Plugins”.

Normally, a client program that supports the use of authentication plugins 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.

Should a client program require lower-level plugin management, the client library contains functions that take an `st_mysql_client_plugin` argument. See C API Client Plugin Interface.

4.4.3 Compiling and Installing Plugin Libraries

After your plugin is written, you must compile it and install it. The procedure for compiling shared objects varies from system to system. If you build your library using `CMake`, it should be able to generate the correct compilation commands for your system. If the library is named `somepluglib`, you should end up with a shared library file that has a name something like `somepluglib.so`. (The `.so` file name suffix might differ on your system.)

To use `CMake`, you'll need to set up the configuration files to enable the plugin to be compiled and installed. Use the plugin examples under the `plugin` directory of a MySQL source distribution as a guide.

Create `CMakeLists.txt`, which should look something like this:

```
MYSQL_ADD_PLUGIN(somepluglib somepluglib.c
  MODULE_ONLY MODULE_OUTPUT_NAME "somepluglib")
```

When `CMake` generates the `Makefile`, it should take care of passing to the compilation command the `-DMYSQL_DYNAMIC_PLUGIN` flag, and passing to the linker the `-lmysqlservices` flag, which is needed to link in any functions from services provided through the plugin services interface. See MySQL Plugin Services.

Run `CMake`, then run `make`:

```
shell> cmake .
shell> make
```

If you need to specify configuration options to `CMake`, see MySQL Source-Configuration Options, for a list. For example, you might want to specify `CMAKE_INSTALL_PREFIX` to indicate the MySQL base directory under which the plugin should be installed. You can see what value to use for this option with `SHOW VARIABLES`:

```
mysql> SHOW VARIABLES LIKE 'basedir';
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| base          | /usr/local/mysql |
+---------------+-----------------+
```

The location of the plugin directory where you should install the library is given by the `plugin_dir` system variable. For example:

```
mysql> SHOW VARIABLES LIKE 'plugin_dir';
+---------------+-----------------------------------+
| Variable_name | Value                             |
+---------------+-----------------------------------+
| plugin_dir    | /usr/local/mysql/lib/mysql/plugin |
+---------------+-----------------------------------+
```

To install the plugin library, use `make`:

```
shell> make install
```

Verify that `make install` installed the plugin library in the proper directory. After installing it, make sure that the library permissions permit it to be executed by the server.

4.4.4 Writing Full-Text Parser Plugins

MySQL supports server-side full-text parser plugins with `MyISAM` and `InnoDB`. For introductory information about full-text parser plugins, see Full-Text Parser Plugins.

A full-text parser plugin can be used to replace or modify the built-in full-text parser. This section describes how to write a full-text parser plugin named `simple_parser`. This plugin performs parsing based on simpler rules than those used by the MySQL built-in full-text parser: Words are nonempty runs of whitespace characters.

The instructions use the source code in the `plugin/fulltext` directory of MySQL source distributions, so change location into that directory. The following procedure describes how the plugin library is created:

1. To write a full-text parser plugin, include the following header file in the plugin source file. Other MySQL or general header files might also be needed, depending on the plugin capabilities and requirements.

   ```
   #include <mysql/plugin.h>
   ```

   `plugin.h` defines the `MYSQL_FTPARSER_PLUGIN` server plugin type and the data structures needed to declare the plugin.

2. Set up the library descriptor for the plugin library file.

   This descriptor contains the general plugin descriptor for the server plugin. For a full-text parser plugin, the type must be `MYSQL_FTPARSER_PLUGIN`. This is the value that identifies the plugin as being legal for use in a `WITH PARSER` clause when creating a `FULLTEXT` index. (No other plugin type is legal for this clause.)

   For example, the library descriptor for a library that contains a single full-text parser plugin named `simple_parser` looks like this:

   ```
   mysql_declare_plugin(ftexample)
   {
     MYSQL_FTPARSER_PLUGIN,      /* type                            */
     &simple_parser_descriptor,  /* descriptor                      */
     "simple_parser",            /* name                            */
   ```

```
  "Oracle Corporation",       /* author                          */
  "Simple Full-Text Parser",  /* description                     */
  PLUGIN_LICENSE_GPL,         /* plugin license                  */
  simple_parser_plugin_init,  /* init function (when loaded)     */
  simple_parser_plugin_deinit,/* deinit function (when unloaded) */
  0x0001,                     /* version                         */
  simple_status,              /* status variables                */
  simple_system_variables,    /* system variables                */
  NULL,
  0
}
mysql_declare_plugin_end;
```

The `name` member (`simple_parser`) indicates the name to use for references to the plugin in statements such as `INSTALL PLUGIN` or `UNINSTALL PLUGIN`. This is also the name displayed by `SHOW PLUGINS` or `INFORMATION_SCHEMA.PLUGINS`.

For more information, see Section 4.4.2.1, “Server Plugin Library and Plugin Descriptors”.

3. Set up the type-specific plugin descriptor.

   Each general plugin descriptor in the library descriptor points to a type-specific descriptor. For a full-text parser plugin, the type-specific descriptor is an instance of the `st_mysql_ftparser` structure in the `plugin.h` file:

   ```
   struct st_mysql_ftparser
   {
     int interface_version;
     int (*parse)(MYSQL_FTPARSER_PARAM *param);
     int (*init)(MYSQL_FTPARSER_PARAM *param);
     int (*deinit)(MYSQL_FTPARSER_PARAM *param);
   };
   ```

   As shown by the structure definition, the descriptor has an interface version number and contains pointers to three functions.

   The interface version number is specified using a symbol, which is in the form: `MYSQL_xxx_INTERFACE_VERSION`. For full-text parser plugins, the symbol is `MYSQL_FTPARSER_INTERFACE_VERSION`. In the source code, you will find the actual interface version number for the full-text parser plugin defined in `include/mysql/plugin_ftparser.h`. With the introduction of full-text parser plugin support for `InnoDB`, the interface version number was incremented in MySQL 5.7 from `0x0100` to `0x0101`.

   The `init` and `deinit` members should point to a function or be set to 0 if the function is not needed. The `parse` member must point to the function that performs the parsing.

   In the `simple_parser` declaration, that descriptor is indicated by `&simple_parser_descriptor`. The descriptor specifies the version number for the full-text plugin interface (as given by `MYSQL_FTPARSER_INTERFACE_VERSION`), and the plugin's parsing, initialization, and deinitialization functions:

   ```
   static struct st_mysql_ftparser simple_parser_descriptor=
   {
     MYSQL_FTPARSER_INTERFACE_VERSION, /* interface version      */
     simple_parser_parse,              /* parsing function       */
     simple_parser_init,               /* parser init function   */
     simple_parser_deinit              /* parser deinit function */
   };
   ```

   A full-text parser plugin is used in two different contexts, indexing and searching. In both contexts, the server calls the initialization and deinitialization functions at the beginning and end of processing each

SQL statement that causes the plugin to be invoked. However, during statement processing, the server calls the main parsing function in context-specific fashion:

- For indexing, the server calls the parser for each column value to be indexed.
- For searching, the server calls the parser to parse the search string. The parser might also be called for rows processed by the statement. In natural language mode, there is no need for the server to call the parser. For boolean mode phrase searches or natural language searches with query expansion, the parser is used to parse column values for information that is not in the index. Also, if a boolean mode search is done for a column that has no `FULLTEXT` index, the built-in parser will be called. (Plugins are associated with specific indexes. If there is no index, no plugin is used.)

The plugin declaration in the general plugin descriptor has `init` and `deinit` members that point initialization and deinitialization functions, and so does the type-specific plugin descriptor to which it points. However, these pairs of functions have different purposes and are invoked for different reasons:

- For the plugin declaration in the general plugin descriptor, the initialization and deinitialization functions are invoked when the plugin is loaded and unloaded.
- For the type-specific plugin descriptor, the initialization and deinitialization functions are invoked per SQL statement for which the plugin is used.

Each interface function named in the plugin descriptor should return zero for success or nonzero for failure, and each of them receives an argument that points to a `MYSQL_FTPARSER_PARAM` structure containing the parsing context. The structure has this definition:

```
typedef struct st_mysql_ftparser_param
{
  int (*mysql_parse)(struct st_mysql_ftparser_param *,
                     char *doc, int doc_len);
  int (*mysql_add_word)(struct st_mysql_ftparser_param *,
                        char *word, int word_len,
                        MYSQL_FTPARSER_BOOLEAN_INFO *boolean_info);
  void *ftparser_state;
  void *mysql_ftparam;
  struct charset_info_st *cs;
  char *doc;
  int length;
  int flags;
  enum enum_ftparser_mode mode;
} MYSQL_FTPARSER_PARAM;
```

The structure members are used as follows:

- `mysql_parse`: A pointer to a callback function that invokes the server's built-in parser. Use this callback when the plugin acts as a front end to the built-in parser. That is, when the plugin parsing function is called, it should process the input to extract the text and pass the text to the `mysql_parse` callback.

  The first parameter for this callback function should be the `param` value itself:

  ```
  param->mysql_parse(param, ...);
  ```

  A front end plugin can extract text and pass it all at once to the built-in parser, or it can extract and pass text to the built-in parser a piece at a time. However, in this case, the built-in parser treats the pieces of text as though there are implicit word breaks between them.

- `mysql_add_word`: A pointer to a callback function that adds a word to a full-text index or to the list of search terms. Use this callback when the parser plugin replaces the built-in parser. That

is, when the plugin parsing function is called, it should parse the input into words and invoke the `mysql_add_word` callback for each word.

The first parameter for this callback function should be the `param` value itself:

```
param->mysql_add_word(param, ...);
```

- `ftparser_state`: This is a generic pointer. The plugin can set it to point to information to be used internally for its own purposes.
- `mysql_ftparam`: This is set by the server. It is passed as the first argument to the `mysql_parse` or `mysql_add_word` callback.
- `cs`: A pointer to information about the character set of the text, or 0 if no information is available.
- `doc`: A pointer to the text to be parsed.
- `length`: The length of the text to be parsed, in bytes.
- `flags`: Parser flags. This is zero if there are no special flags. The only nonzero flag is `MYSQL_FTFLAGS_NEED_COPY`, which means that `mysql_add_word()` must save a copy of the word (that is, it cannot use a pointer to the word because the word is in a buffer that will be overwritten.)

  This flag might be set or reset by MySQL before calling the parser plugin, by the parser plugin itself, or by the `mysql_parse()` function.
- `mode`: The parsing mode. This value will be one of the following constants:
  - `MYSQL_FTPARSER_SIMPLE_MODE`: Parse in fast and simple mode, which is used for indexing and for natural language queries. The parser should pass to the server only those words that should be indexed. If the parser uses length limits or a stopword list to determine which words to ignore, it should not pass such words to the server.
  - `MYSQL_FTPARSER_WITH_STOPWORDS`: Parse in stopword mode. This is used in boolean searches for phrase matching. The parser should pass all words to the server, even stopwords or words that are outside any normal length limits.
  - `MYSQL_FTPARSER_FULL_BOOLEAN_INFO`: Parse in boolean mode. This is used for parsing boolean query strings. The parser should recognize not only words but also boolean-mode operators and pass them to the server as tokens using the `mysql_add_word` callback. To tell the server what kind of token is being passed, the plugin needs to fill in a `MYSQL_FTPARSER_BOOLEAN_INFO` structure and pass a pointer to it.

    > **Note**
    >
    > For `MyISAM`, the stopword list and `ft_min_word_len` and `ft_max_word_len` are checked inside the tokenizer. For `InnoDB`, the stopword list and equivalent word length variable settings (`innodb_ft_min_token_size` and `innodb_ft_max_token_size`) are checked outside of the tokenizer. As a result, `InnoDB` plugin parsers do not need to check the stopword list, `innodb_ft_min_token_size`, or `innodb_ft_max_token_size`. Instead, it is recommended that all words be returned to `InnoDB`. However, if you want to check stopwords within your plugin parser, use `MYSQL_FTPARSER_SIMPLE_MODE`, which is for full-text search index and natural language search. For `MYSQL_FTPARSER_WITH_STOPWORDS`

> and `MYSQL_FTPARSER_FULL_BOOLEAN_INFO` modes, it is recommended that all words be returned to `InnoDB` including stopwords, in case of phrase searches.

If the parser is called in boolean mode, the `param->mode` value will be `MYSQL_FTPARSER_FULL_BOOLEAN_INFO`. The `MYSQL_FTPARSER_BOOLEAN_INFO` structure that the parser uses for passing token information to the server looks like this:

```
typedef struct st_mysql_ftparser_boolean_info
{
  enum enum_ft_token_type type;
  int yesno;
  int weight_adjust;
  char wasign;
  char trunc;
  int position;
  /* These are parser state and must be removed. */
  char prev;
  char *quot;
} MYSQL_FTPARSER_BOOLEAN_INFO;
```

The parser should fill in the structure members as follows:

- `type`: The token type. The following table shows the permissible types.

Table 4.3 Full-Text Parser Token Types

Token Value	Meaning
`FT_TOKEN_EOF`	End of data
`FT_TOKEN_WORD`	A regular word
`FT_TOKEN_LEFT_PAREN`	The beginning of a group or subexpression
`FT_TOKEN_RIGHT_PAREN`	The end of a group or subexpression
`FT_TOKEN_STOPWORD`	A stopword

- `yesno`: Whether the word must be present for a match to occur. 0 means that the word is optional but increases the match relevance if it is present. Values larger than 0 mean that the word must be present. Values smaller than 0 mean that the word must not be present.
- `weight_adjust`: A weighting factor that determines how much a match for the word counts. It can be used to increase or decrease the word's importance in relevance calculations. A value of zero indicates no weight adjustment. Values greater than or less than zero mean higher or lower weight, respectively. The examples at Boolean Full-Text Searches, that use the `<` and `>` operators illustrate how weighting works.
- `wasign`: The sign of the weighting factor. A negative value acts like the `~` boolean-search operator, which causes the word's contribution to the relevance to be negative.
- `trunc`: Whether matching should be done as if the boolean-mode `*` truncation operator had been given.

- `position`: Start position of the word in the document, in bytes. Used by `InnoDB` full-text search. For existing plugins that are called in boolean mode, support must be added for the position member.

Plugins should not use the `prev` and `quot` members of the `MYSQL_FTPARSER_BOOLEAN_INFO` structure.

> **Note**
>
> The plugin parser framework does not support:
>
> - The `@distance` boolean operator.
> - A leading plus sign (`+`) or minus sign (`-`) boolean operator followed by a space and then a word (`'+ apple'` or `'- apple'`). The leading plus or minus sign must be directly adjacent to the word, for example: `'+apple'` or `'-apple'`.
>
> For information about boolean full-text search operators, see Boolean Full-Text Searches.

4. Set up the plugin interface functions.

   The general plugin descriptor in the library descriptor names the initialization and deinitialization functions that the server should invoke when it loads and unloads the plugin. For `simple_parser`, these functions do nothing but return zero to indicate that they succeeded:

```
static int simple_parser_plugin_init(void *arg __attribute__((unused)))
{
  return(0);
}

static int simple_parser_plugin_deinit(void *arg __attribute__((unused)))
{
  return(0);
}
```

   Because those functions do not actually do anything, you could omit them and specify 0 for each of them in the plugin declaration.

   The type-specific plugin descriptor for `simple_parser` names the initialization, deinitialization, and parsing functions that the server invokes when the plugin is used. For `simple_parser`, the initialization and deinitialization functions do nothing:

```
static int simple_parser_init(MYSQL_FTPARSER_PARAM *param
                              __attribute__((unused)))
{
  return(0);
}

static int simple_parser_deinit(MYSQL_FTPARSER_PARAM *param
                                __attribute__((unused)))
{
  return(0);
```

```
}
```

Here too, because those functions do nothing, you could omit them and specify 0 for each of them in the plugin descriptor.

The main parsing function, `simple_parser_parse()`, acts as a replacement for the built-in full-text parser, so it needs to split text into words and pass each word to the server. The parsing function's first argument is a pointer to a structure that contains the parsing context. This structure has a `doc` member that points to the text to be parsed, and a `length` member that indicates how long the text is. The simple parsing done by the plugin considers nonempty runs of whitespace characters to be words, so it identifies words like this:

```
static int simple_parser_parse(MYSQL_FTPARSER_PARAM *param)
{
  char *end, *start, *docend= param->doc + param->length;

  for (end= start= param->doc;; end++)
  {
    if (end == docend)
    {
      if (end > start)
        add_word(param, start, end - start);
      break;
    }
    else if (isspace(*end))
    {
      if (end > start)
        add_word(param, start, end - start);
      start= end + 1;
    }
  }
  return(0);
}
```

As the parser finds each word, it invokes a function `add_word()` to pass the word to the server. `add_word()` is a helper function only; it is not part of the plugin interface. The parser passes the parsing context pointer to `add_word()`, as well as a pointer to the word and a length value:

```
static void add_word(MYSQL_FTPARSER_PARAM *param, char *word, size_t len)
{
  MYSQL_FTPARSER_BOOLEAN_INFO bool_info=
    { FT_TOKEN_WORD, 0, 0, 0, 0, 0, ' ', 0 };

  param->mysql_add_word(param, word, len, &bool_info);
}
```

For boolean-mode parsing, `add_word()` fills in the members of the `bool_info` structure as described earlier in the discussion of the `st_mysql_ftparser_boolean_info` structure.

5. Set up the status variables. For the `simple_parser` plugin, the following status variable array sets up one status variable with a value that is static text, and another with a value that is stored in a long integer variable:

```
long number_of_calls= 0;

struct st_mysql_show_var simple_status[]=
{
  {"simple_parser_static", (char *)"just a static text", SHOW_CHAR},
  {"simple_parser_called", (char *)&number_of_calls,     SHOW_LONG},
  {0,0,0}
```

```
};
```

By using status variable names that begin with the plugin name, you can easily display the variables for a plugin with `SHOW STATUS`:

```
mysql> SHOW STATUS LIKE 'simple_parser%';
+----------------------+--------------------+
| Variable_name        | Value              |
+----------------------+--------------------+
| simple_parser_static | just a static text |
| simple_parser_called | 0                  |
+----------------------+--------------------+
```

6. To compile and install a plugin library file, use the instructions in Section 4.4.3, “Compiling and Installing Plugin Libraries”. To make the library file available for use, install it in the plugin directory (the directory named by the `plugin_dir` system variable). For the `simple_parser` plugin, it is compiled and installed when you build MySQL from source. It is also included in binary distributions. The build process produces a shared object library with a name of `mypluglib.so` (the `.so` suffix might differ depending on your platform).

7. To use the plugin, register it with the server. For example, to register the plugin at runtime, use this statement, adjusting the `.so` suffix for your platform as necessary:

   ```
   INSTALL PLUGIN simple_parser SONAME 'mypluglib.so';
   ```

   For additional information about plugin loading, see Installing and Uninstalling Plugins.

8. To verify plugin installation, examine the `INFORMATION_SCHEMA.PLUGINS` table or use the `SHOW PLUGINS` statement. See Obtaining Server Plugin Information.

9. Test the plugin to verify that it works properly.

   Create a table that contains a string column and associate the parser plugin with a `FULLTEXT` index on the column:

   ```
   mysql> CREATE TABLE t (c VARCHAR(255),
       ->   FULLTEXT (c) WITH PARSER simple_parser
       -> ) ENGINE=MyISAM;
   Query OK, 0 rows affected (0.01 sec)
   ```

   Insert some text into the table and try some searches. These should verify that the parser plugin treats all nonwhitespace characters as word characters:

   ```
   mysql> INSERT INTO t VALUES
       ->   ('latin1_general_cs is a case-sensitive collation'),
       ->   ('I\'d like a case of oranges'),
       ->   ('this is sensitive information'),
       ->   ('another row'),
       ->   ('yet another row');
   Query OK, 5 rows affected (0.02 sec)
   Records: 5  Duplicates: 0  Warnings: 0

   mysql> SELECT c FROM t;

   +-----------------------------------------------+
   | c                                             |
   +-----------------------------------------------+
   | latin1_general_cs is a case-sensitive collation |
   | I'd like a case of oranges                    |
   | this is sensitive information                 |
   | another row                                   |
   | yet another row                               |
   +-----------------------------------------------+
   ```

```
5 rows in set (0.00 sec)

mysql> SELECT MATCH(c) AGAINST('case') FROM t;
+-------------------------+
| MATCH(c) AGAINST('case') |
+-------------------------+
|                       0 |
|         1.2968142032623 |
|                       0 |
|                       0 |
|                       0 |
+-------------------------+
5 rows in set (0.00 sec)

mysql> SELECT MATCH(c) AGAINST('sensitive') FROM t;
+-----------------------------+
| MATCH(c) AGAINST('sensitive') |
+-----------------------------+
|                           0 |
|                           0 |
|              1.3253291845322 |
|                           0 |
|                           0 |
+-----------------------------+
5 rows in set (0.01 sec)

mysql> SELECT MATCH(c) AGAINST('case-sensitive') FROM t;
+----------------------------------+
| MATCH(c) AGAINST('case-sensitive') |
+----------------------------------+
|                  1.3109166622162 |
|                                0 |
|                                0 |
|                                0 |
|                                0 |
+----------------------------------+
5 rows in set (0.01 sec)

mysql> SELECT MATCH(c) AGAINST('I\'d') FROM t;
+-------------------------+
| MATCH(c) AGAINST('I\'d') |
+-------------------------+
|                       0 |
|         1.2968142032623 |
|                       0 |
|                       0 |
|                       0 |
+-------------------------+
5 rows in set (0.01 sec)
```

Neither “case” nor “insensitive” match “case-insensitive” the way that they would for the built-in parser.

4.4.5 Writing Daemon Plugins

A daemon plugin is a simple type of plugin used for code that should be run by the server but that does not communicate with it. This section describes how to write a daemon server plugin, using the example plugin found in the `plugin/daemon_example` directory of MySQL source distributions. That directory contains the `daemon_example.cc` source file for a daemon plugin named `daemon_example` that writes a heartbeat string at regular intervals to a file named `mysql-heartbeat.log` in the data directory.

To write a daemon plugin, include the following header file in the plugin source file. Other MySQL or general header files might also be needed, depending on the plugin capabilities and requirements.

```
#include <mysql/plugin.h>
```

`plugin.h` defines the `MYSQL_DAEMON_PLUGIN` server plugin type and the data structures needed to declare the plugin.

The `daemon_example.cc` file sets up the library descriptor as follows. The library descriptor includes a single general server plugin descriptor.

```
mysql_declare_plugin(daemon_example)
{
  MYSQL_DAEMON_PLUGIN,
  &daemon_example_plugin,
  "daemon_example",
  "Brian Aker",
  "Daemon example, creates a heartbeat beat file in mysql-heartbeat.log",
  PLUGIN_LICENSE_GPL,
  daemon_example_plugin_init, /* Plugin Init */
  daemon_example_plugin_deinit, /* Plugin Deinit */
  0x0100 /* 1.0 */,
  NULL,                       /* status variables                */
  NULL,                       /* system variables                */
  NULL,                       /* config options                  */
  0,                          /* flags                           */
}
mysql_declare_plugin_end;
```

The `name` member (`daemon_example`) indicates the name to use for references to the plugin in statements such as `INSTALL PLUGIN` or `UNINSTALL PLUGIN`. This is also the name displayed by `SHOW PLUGINS` or `INFORMATION_SCHEMA.PLUGINS`.

The second member of the plugin descriptor, `daemon_example_plugin`, points to the type-specific daemon plugin descriptor. This structure consists only of the type-specific API version number:

```
struct st_mysql_daemon daemon_example_plugin=
{ MYSQL_DAEMON_INTERFACE_VERSION  };
```

The type-specific structure has no interface functions. There is no communication between the server and the plugin, except that the server calls the initialization and deinitialization functions from the general plugin descriptor to start and stop the plugin:

- `daemon_example_plugin_init()` opens the heartbeat file and spawns a thread that wakes up periodically and writes the next message to the file.
- `daemon_example_plugin_deinit()` closes the file and performs other cleanup.

To compile and install a plugin library file, use the instructions in Section 4.4.3, “Compiling and Installing Plugin Libraries”. To make the library file available for use, install it in the plugin directory (the directory named by the `plugin_dir` system variable). For the `daemon_example` plugin, it is compiled and installed when you build MySQL from source. It is also included in binary distributions. The build process produces a shared object library with a name of `libdaemon_example.so` (the `.so` suffix might differ depending on your platform).

To use the plugin, register it with the server. For example, to register the plugin at runtime, use this statement, adjusting the `.so` suffix for your platform as necessary:

```
INSTALL PLUGIN daemon_example SONAME 'libdaemon_example.so';
```

For additional information about plugin loading, see Installing and Uninstalling Plugins.

To verify plugin installation, examine the `INFORMATION_SCHEMA.PLUGINS` table or use the `SHOW PLUGINS` statement. See Obtaining Server Plugin Information.

While the plugin is loaded, it writes a heartbeat string at regular intervals to a file named `mysql-heartbeat.log` in the data directory. This file grows without limit, so after you have satistifed yourself that the plugin operates correctly, unload it:

```
UNINSTALL PLUGIN daemon_example;
```

4.4.6 Writing INFORMATION_SCHEMA Plugins

This section describes how to write a server-side `INFORMATION_SCHEMA` table plugin. For example code that implements such plugins, see the `sql/sql_show.cc` file of a MySQL source distribution. You can also look at the example plugins found in the `InnoDB` source. See the `handler/i_s.cc` and `handler/ha_innodb.cc` files within the `InnoDB` source tree (in the `storage/innobase` directory).

To write an `INFORMATION_SCHEMA` table plugin, include the following header files in the plugin source file. Other MySQL or general header files might also be needed, depending on the plugin capabilities and requirements.

```
#include <sql_class.h>
#include <table.h>
```

These header files are located in the `sql` directory of MySQL source distributions. They contain C++ structures, so the source file for an `INFORMATION_SCHEMA` plugin must be compiled as C++ (not C) code.

The source file for the example plugin developed here is named `simple_i_s_table.cc`. It creates a simple `INFORMATION_SCHEMA` table named `SIMPLE_I_S_TABLE` that has two columns named `NAME` and `VALUE`. The general descriptor for a plugin library that implements the table looks like this:

```
mysql_declare_plugin(simple_i_s_library)
{
  MYSQL_INFORMATION_SCHEMA_PLUGIN,
  &simple_table_info,                /* type-specific descriptor */
  "SIMPLE_I_S_TABLE",                /* table name */
  "Author Name",                     /* author */
  "Simple INFORMATION_SCHEMA table", /* description */
  PLUGIN_LICENSE_GPL,                /* license type */
  simple_table_init,                 /* init function */
  NULL,
  0x0100,                            /* version = 1.0 */
  NULL,                              /* no status variables */
  NULL,                              /* no system variables */
  NULL,                              /* no reserved information */
  0                                  /* no flags */
}
mysql_declare_plugin_end;
```

The `name` member (`SIMPLE_I_S_TABLE`) indicates the name to use for references to the plugin in statements such as `INSTALL PLUGIN` or `UNINSTALL PLUGIN`. This is also the name displayed by `SHOW PLUGINS` or `INFORMATION_SCHEMA.PLUGINS`.

The `simple_table_info` member of the general descriptor points to the type-specific descriptor, which consists only of the type-specific API version number:

```
static struct st_mysql_information_schema simple_table_info =
{ MYSQL_INFORMATION_SCHEMA_INTERFACE_VERSION };
```

The general descriptor points to the initialization and deinitialization functions:

- The initialization function provides information about the table structure and a function that populates the table.

- The deinitialization function performs any required cleanup. If no cleanup is needed, this descriptor member can be `NULL` (as in the example shown).

The initialization function should return 0 for success, 1 if an error occurs. The function receives a generic pointer, which it should interpret as a pointer to the table structure:

```
static int table_init(void *ptr)
{
  ST_SCHEMA_TABLE *schema_table= (ST_SCHEMA_TABLE*)ptr;

  schema_table->fields_info= simple_table_fields;
  schema_table->fill_table= simple_fill_table;
  return 0;
}
```

The function should set these two members of the table structure:

- `fields_info`: An array of `ST_FIELD_INFO` structures that contain information about each column.
- `fill_table`: A function that populates the table.

The array pointed to by `fields_info` should contain one element per column of the `INFORMATION_SCHEMA` plus a terminating element. The following `simple_table_fields` array for the example plugin indicates that `SIMPLE_I_S_TABLE` has two columns. `NAME` is string-valued with a length of 10 and `VALUE` is integer-valued with a display width of 20. The last structure marks the end of the array.

```
static ST_FIELD_INFO simple_table_fields[]=
{
  {"NAME", 10, MYSQL_TYPE_STRING, 0, 0 0, 0},
  {"VALUE", 6, MYSQL_TYPE_LONG, 0, MY_I_S_UNSIGNED, 0, 0},
  {0, 0, MYSQL_TYPE_NULL, 0, 0, 0, 0}
};
```

For more information about the column information structure, see the definition of `ST_FIELD_INFO` in the `table.h` header file. The permissible `MYSQL_TYPE_`*`xxx`* type values are those used in the C API; see C API Basic Data Structures.

The `fill_table` member should be set to a function that populates the table and returns 0 for success, 1 if an error occurs. For the example plugin, the `simple_fill_table()` function looks like this:

```
static int simple_fill_table(THD *thd, TABLE_LIST *tables, Item *cond)
{
  TABLE *table= tables->table;

  table->field[0]->store("Name 1", 6, system_charset_info);
  table->field[1]->store(1);
  if (schema_table_store_record(thd, table))
    return 1;
  table->field[0]->store("Name 2", 6, system_charset_info);
  table->field[1]->store(2);
  if (schema_table_store_record(thd, table))
    return 1;
  return 0;
}
```

For each row of the `INFORMATION_SCHEMA` table, this function initializes each column, then calls `schema_table_store_record()` to install the row. The `store()` method arguments depend on the type of value to be stored. For column 0 (`NAME`, a string), `store()` takes a pointer to a string, its length, and information about the character set of the string:

```
store(const char *to, uint length, CHARSET_INFO *cs);
```

For column 1 (`VALUE`, an integer), `store()` takes the value and a flag indicating whether it is unsigned:

```
store(longlong nr, bool unsigned_value);
```

For other examples of how to populate `INFORMATION_SCHEMA` tables, search for instances of `schema_table_store_record()` in `sql_show.cc`.

To compile and install a plugin library file, use the instructions in Section 4.4.3, “Compiling and Installing Plugin Libraries”. To make the library file available for use, install it in the plugin directory (the directory named by the `plugin_dir` system variable).

To test the plugin, install it:

```
mysql> INSTALL PLUGIN SIMPLE_I_S_TABLE SONAME 'simple_i_s_table.so';
```

Verify that the table is present:

```
mysql> SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_NAME = 'SIMPLE_I_S_TABLE';
+------------------+
| TABLE_NAME       |
+------------------+
| SIMPLE_I_S_TABLE |
+------------------+
```

Try to select from it:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.SIMPLE_I_S_TABLE;
+--------+-------+
| NAME   | VALUE |
+--------+-------+
| Name 1 |     1 |
| Name 2 |     2 |
+--------+-------+
```

Uninstall it:

```
mysql> UNINSTALL PLUGIN SIMPLE_I_S_TABLE;
```

4.4.7 Writing Semisynchronous Replication Plugins

This section describes how to write server-side semisynchronous replication plugins, using the example plugins found in the `plugin/semisync` directory of MySQL source distributions. That directory contains the source files for source and replica plugins named `rpl_semi_sync_master` and `rpl_semi_sync_slave`. The information here covers only how to set up the plugin framework. For details about how the plugins implement replication functions, see the source.

To write a semisynchronous replication plugin, include the following header file in the plugin source file. Other MySQL or general header files might also be needed, depending on the plugin capabilities and requirements.

```
#include <mysql/plugin.h>
```

`plugin.h` defines the `MYSQL_REPLICATION_PLUGIN` server plugin type and the data structures needed to declare the plugin.

For the source side, `semisync_master_plugin.cc` contains this general descriptor for a plugin named `rpl_semi_sync_master`:

```
mysql_declare_plugin(semi_sync_master)
{
```

```
  MYSQL_REPLICATION_PLUGIN,
  &semi_sync_master_plugin,
  "rpl_semi_sync_master",
  "He Zhenxing",
  "Semi-synchronous replication master",
  PLUGIN_LICENSE_GPL,
  semi_sync_master_plugin_init, /* Plugin Init */
  semi_sync_master_plugin_deinit, /* Plugin Deinit */
  0x0100 /* 1.0 */,
  semi_sync_master_status_vars, /* status variables */
  semi_sync_master_system_vars, /* system variables */
  NULL,                         /* config options */
  0,                            /* flags */
}
mysql_declare_plugin_end;
```

For the replica side, `semisync_slave_plugin.cc` contains this general descriptor for a plugin named `rpl_semi_sync_slave`:

```
mysql_declare_plugin(semi_sync_slave)
{
  MYSQL_REPLICATION_PLUGIN,
  &semi_sync_slave_plugin,
  "rpl_semi_sync_slave",
  "He Zhenxing",
  "Semi-synchronous replication slave",
  PLUGIN_LICENSE_GPL,
  semi_sync_slave_plugin_init, /* Plugin Init */
  semi_sync_slave_plugin_deinit, /* Plugin Deinit */
  0x0100 /* 1.0 */,
  semi_sync_slave_status_vars,  /* status variables */
  semi_sync_slave_system_vars,  /* system variables */
  NULL,                         /* config options */
  0,                            /* flags */
}
mysql_declare_plugin_end;
```

For both the source and replica plugins, the general descriptor has pointers to the type-specific descriptor, the initialization and deinitialization functions, and to the status and system variables implemented by the plugin. For information about variable setup, see Section 4.4.2.2, “Server Plugin Status and System Variables”. The following remarks discuss the type-specific descriptor and the initialization and deinitialization functions for the source plugin but apply similarly to the replica plugin.

The `semi_sync_master_plugin` member of the source general descriptor points to the type-specific descriptor, which consists only of the type-specific API version number:

```
struct Mysql_replication semi_sync_master_plugin= {
  MYSQL_REPLICATION_INTERFACE_VERSION
};
```

The initialization and deinitialization function declarations look like this:

```
static int semi_sync_master_plugin_init(void *p);
static int semi_sync_master_plugin_deinit(void *p);
```

The initialization function uses the pointer to register transaction and binary logging “observers” with the server. After successful initialization, the server takes care of invoking the observers at the appropriate times. (For details on the observers, see the source files.) The deinitialization function cleans up by deregistering the observers. Each function returns 0 for success or 1 if an error occurs.

To compile and install a plugin library file, use the instructions in Section 4.4.3, “Compiling and Installing Plugin Libraries”. To make the library file available for use, install it in the plugin directory (the directory named by the `plugin_dir` system variable). For the `rpl_semi_sync_master` and

`rpl_semi_sync_slave` plugins, they are compiled and installed when you build MySQL from source. They are also included in binary distributions. The build process produces shared object libraries with names of `semisync_master.so` and `semisync_slave.so` (the `.so` suffix might differ depending on your platform).

4.4.8 Writing Audit Plugins

This section describes how to write a server-side audit plugin, using the example plugin found in the `plugin/audit_null` directory of MySQL source distributions. The `audit_null.c` and `audit_null_variables.h` source files in that directory implement an audit plugin named `NULL_AUDIT`.

> **Note**
>
> Changes were made in MySQL 5.7 to reimplement query rewrite plugins as audit plugins, and the audit plugin API itself was extensively revised in comparison to MySQL 5.6. To write audit plugins against the older API, see Writing Audit Plugins in Extending MySQL 5.6.

> **Note**
>
> Other examples of plugins that use the audit plugin API are the query rewrite plugin (see The Rewriter Query Rewrite Plugin) and the Version Tokens plugin (see Version Tokens).

Within the server, the pluggable audit interface is implemented in the `sql_audit.h` and `sql_audit.cc` files in the `sql` directory of MySQL source distributions. Additionally, several places in the server call the audit interface when an auditable event occurs, so that registered audit plugins can be notified about the event if necessary. To see where such calls occur, search the server source files for invocations of functions with names of the form `mysql_audit_xxx()`. Audit notification occurs for server operations such as these:

- Client connect and disconnect events
- Writing a message to the general query log (if the log is enabled)
- Writing a message to the error log
- Sending a query result to a client

To write an audit plugin, include the following header file in the plugin source file. Other MySQL or general header files might also be needed, depending on the plugin capabilities and requirements.

```
#include <mysql/plugin_audit.h>
```

`plugin_audit.h` includes `plugin.h`, so you need not include the latter file explicitly. `plugin.h` defines the `MYSQL_AUDIT_PLUGIN` server plugin type and the data structures needed to declare the plugin. `plugin_audit.h` defines data structures specific to audit plugins.

- Audit Plugin General Descriptor
- Audit Plugin Type-Specific Descriptor
- Audit Plugin Notification Function
- Audit Plugin Error Handling
- Audit Plugin Usage

Audit Plugin General Descriptor

An audit plugin, like any MySQL server plugin, has a general plugin descriptor (see Section 4.4.2.1, "Server Plugin Library and Plugin Descriptors") and a type-specific plugin descriptor. In `audit_null.c`, the general descriptor for `audit_null` looks like this:

```
mysql_declare_plugin(audit_null)
{
  MYSQL_AUDIT_PLUGIN,         /* type                            */
  &audit_null_descriptor,     /* descriptor                      */
  "NULL_AUDIT",               /* name                            */
  "Oracle Corp",              /* author                          */
  "Simple NULL Audit",        /* description                     */
  PLUGIN_LICENSE_GPL,
  audit_null_plugin_init,     /* init function (when loaded)     */
  audit_null_plugin_deinit,   /* deinit function (when unloaded) */
  0x0003,                     /* version                         */
  simple_status,              /* status variables                */
  system_variables,           /* system variables                */
  NULL,
  0,
}
mysql_declare_plugin_end;
```

The first member, `MYSQL_AUDIT_PLUGIN`, identifies this plugin as an audit plugin.

`audit_null_descriptor` points to the type-specific plugin descriptor, described later.

The `name` member (`NULL_AUDIT`) indicates the name to use for references to the plugin in statements such as `INSTALL PLUGIN` or `UNINSTALL PLUGIN`. This is also the name displayed by `INFORMATION_SCHEMA.PLUGINS` or `SHOW PLUGINS`.

The `audit_null_plugin_init` initialization function performs plugin initialization when the plugin is loaded. The `audit_null_plugin_deinit` function performs cleanup when the plugin is unloaded.

The general plugin descriptor also refers to `simple_status` and `system_variables`, structures that expose several status and system variables. When the plugin is enabled, these variables can be inspected using `SHOW` statements (`SHOW STATUS`, `SHOW VARIABLES`) or the appropriate Performance Schema tables.

The `simple_status` structure declares several status variables with names of the form `Audit_null_`*`xxx`*. `NULL_AUDIT` increments the `Audit_null_called` status variable for every notification that it receives. The other status variables are more specific and `NULL_AUDIT` increments them only for notifications of specific events.

`system_variables` is an array of system variable elements, each of which is defined using a `MYSQL_THDVAR_`*`xxx`* macro. These system variables have names of the form `null_audit_`*`xxx`*. These variables can be used to communicate with the plugin at runtime.

Audit Plugin Type-Specific Descriptor

The `audit_null_descriptor` value in the general plugin descriptor points to the type-specific plugin descriptor. For audit plugins, this descriptor has the following structure (defined in `plugin_audit.h`):

```
struct st_mysql_audit
{
  int interface_version;
  void (*release_thd)(MYSQL_THD);
  int (*event_notify)(MYSQL_THD, mysql_event_class_t, const void *);
  unsigned long class_mask[MYSQL_AUDIT_CLASS_MASK_SIZE];
};
```

The type-specific descriptor for audit plugins has these members:

- `interface_version`: By convention, type-specific plugin descriptors begin with the interface version for the given plugin type. The server checks `interface_version` when it loads the plugin to see whether the plugin is compatible with it. For audit plugins, the value of the `interface_version` member is `MYSQL_AUDIT_INTERFACE_VERSION` (defined in `plugin_audit.h`).
- `release_thd`: A function that the server calls to inform the plugin that it is being dissociated from its thread context. This should be `NULL` if there is no such function.
- `event_notify`: A function that the server calls to notify the plugin that an auditable event has occurred. This function should not be `NULL`; that would not make sense because no auditing would occur.
- `class_mask`: An array of `MYSQL_AUDIT_CLASS_MASK_SIZE` elements. Each element specifies a bitmask for a given event class to indicate the subclasses for which the plugin wants notification. (This is how the plugin “subscribes” to events of interest.) An element should be 0 to ignore all events for the corresponding event class.

The server uses the `event_notify` and `release_thd` functions together. They are called within the context of a specific thread, and a thread might perform an activity that produces several event notifications. The first time the server calls `event_notify` for a thread, it creates a binding of the plugin to the thread. The plugin cannot be uninstalled while this binding exists. When no more events for the thread will occur, the server informs the plugin of this by calling the `release_thd` function, and then destroys the binding. For example, when a client issues a statement, the thread processing the statement might notify audit plugins about the result set produced by the statement and about the statement being logged. After these notifications occur, the server releases the plugin before putting the thread to sleep until the client issues another statement.

This design enables the plugin to allocate resources needed for a given thread in the first call to the `event_notify` function and release them in the `release_thd` function:

```
event_notify function:
  if memory is needed to service the thread
    allocate memory
  ... rest of notification processing ...

release_thd function:
  if memory was allocated
    release memory
  ... rest of release processing ...
```

That is more efficient than allocating and releasing memory repeatedly in the notification function.

For the `NULL_AUDIT` audit plugin, the type-specific plugin descriptor looks like this:

```
static struct st_mysql_audit audit_null_descriptor=
{
  MYSQL_AUDIT_INTERFACE_VERSION,                  /* interface version    */
  NULL,                                           /* release_thd function */
  audit_null_notify,                              /* notify function      */
  { (unsigned long) MYSQL_AUDIT_GENERAL_ALL,
    (unsigned long) MYSQL_AUDIT_CONNECTION_ALL,
    (unsigned long) MYSQL_AUDIT_PARSE_ALL,
    (unsigned long) MYSQL_AUDIT_AUTHORIZATION_ALL,
    (unsigned long) MYSQL_AUDIT_TABLE_ACCESS_ALL,
    (unsigned long) MYSQL_AUDIT_GLOBAL_VARIABLE_ALL,
    (unsigned long) MYSQL_AUDIT_SERVER_STARTUP_ALL,
    (unsigned long) MYSQL_AUDIT_SERVER_SHUTDOWN_ALL,
    (unsigned long) MYSQL_AUDIT_COMMAND_ALL,
    (unsigned long) MYSQL_AUDIT_QUERY_ALL,
    (unsigned long) MYSQL_AUDIT_STORED_PROGRAM_ALL }
};
```

The server calls `audit_null_notify()` to pass audit event information to the plugin. The plugin has no `release_thd` function.

The `class_mask` member is an array that indicates which event classes the plugin subscribes to. As shown, the array contents subscribe to all subclasses of all event classes that are available. To ignore all notifications for a given event class, specify the corresponding `class_mask` element as 0.

The number of `class_mask` elements corresponds to the number of event classes, each of which is listed in the `mysql_event_class_t` enumeration defined in `plugin_audit.h`:

```
typedef enum
{
  MYSQL_AUDIT_GENERAL_CLASS          = 0,
  MYSQL_AUDIT_CONNECTION_CLASS       = 1,
  MYSQL_AUDIT_PARSE_CLASS            = 2,
  MYSQL_AUDIT_AUTHORIZATION_CLASS    = 3,
  MYSQL_AUDIT_TABLE_ACCESS_CLASS     = 4,
  MYSQL_AUDIT_GLOBAL_VARIABLE_CLASS  = 5,
  MYSQL_AUDIT_SERVER_STARTUP_CLASS   = 6,
  MYSQL_AUDIT_SERVER_SHUTDOWN_CLASS  = 7,
  MYSQL_AUDIT_COMMAND_CLASS          = 8,
  MYSQL_AUDIT_QUERY_CLASS            = 9,
  MYSQL_AUDIT_STORED_PROGRAM_CLASS   = 10,
  /* This item must be last in the list. */
  MYSQL_AUDIT_CLASS_MASK_SIZE
} mysql_event_class_t;
```

For any given event class, `plugin_audit.h` defines bitmask symbols for individual event subclasses, as well as an *`xxx`*`_ALL` symbol that is the union of the all subclass bitmasks. For example, for `MYSQL_AUDIT_CONNECTION_CLASS` (the class that covers connect and disconnect events), `plugin_audit.h` defines these symbols:

```
typedef enum
{
  /** occurs after authentication phase is completed. */
  MYSQL_AUDIT_CONNECTION_CONNECT          = 1 << 0,
  /** occurs after connection is terminated. */
  MYSQL_AUDIT_CONNECTION_DISCONNECT       = 1 << 1,
  /** occurs after COM_CHANGE_USER RPC is completed. */
  MYSQL_AUDIT_CONNECTION_CHANGE_USER      = 1 << 2,
  /** occurs before authentication. */
  MYSQL_AUDIT_CONNECTION_PRE_AUTHENTICATE = 1 << 3
} mysql_event_connection_subclass_t;

#define MYSQL_AUDIT_CONNECTION_ALL (MYSQL_AUDIT_CONNECTION_CONNECT | \
                                    MYSQL_AUDIT_CONNECTION_DISCONNECT | \
                                    MYSQL_AUDIT_CONNECTION_CHANGE_USER | \
                                    MYSQL_AUDIT_CONNECTION_PRE_AUTHENTICATE)
```

To subscribe to all subclasses of the connection event class (as the `NULL_AUDIT` plugin does), a plugin specifies `MYSQL_AUDIT_CONNECTION_ALL` in the corresponding `class_mask` element (`class_mask[1]` in this case). To subscribe to only some subclasses, the plugin sets the `class_mask` element to the union of the subclasses of interest. For example, to subscribe only to the connect and change-user subclasses, the plugin sets `class_mask[1]` to this value:

```
MYSQL_AUDIT_CONNECTION_CONNECT | MYSQL_AUDIT_CONNECTION_CHANGE_USER
```

Audit Plugin Notification Function

Most of the work for an audit plugin occurs in the notification function (the `event_notify` member of the type-specific plugin descriptor). The server calls this function for each auditable event. Audit plugin notification functions have this prototype:

```
int (*event_notify)(MYSQL_THD, mysql_event_class_t, const void *);
```

The second and third parameters of the `event_notify` function prototype represent the event class and a generic pointer to an event structure. (Events in different classes have different structures. The notification function can use the event class value to determine which event structure applies.) The function processes the event and returns a status indicating whether the server should continue processing the event or terminate it.

For `NULL_AUDIT`, the notification function is `audit_null_notify()`. This function increments a global event counter (which the plugin exposes as the value of the `Audit_null_called` status value), and then examines the event class to determine how to process the event structure:

```
static int audit_null_notify(MYSQL_THD thd __attribute__((unused)),
                             mysql_event_class_t event_class,
                             const void *event)
{
  ...

  number_of_calls++;

  if (event_class == MYSQL_AUDIT_GENERAL_CLASS)
  {
    const struct mysql_event_general *event_general=
                                    (const struct mysql_event_general *)event;
    ...
  }
  else if (event_class == MYSQL_AUDIT_CONNECTION_CLASS)
  {
    const struct mysql_event_connection *event_connection=
                                (const struct mysql_event_connection *) event;
    ...

  }
  else if (event_class == MYSQL_AUDIT_PARSE_CLASS)
  {
    const struct mysql_event_parse *event_parse =
                                      (const struct mysql_event_parse *)event;
    ...
  }
  ...
}
```

The notification function interprets the `event` argument according to the value of `event_class`. The `event` argument is a generic pointer to the event record, the structure of which differs per event class. (The `plugin_audit.h` file contains the structures that define the contents of each event class.) For each class, `audit_null_notify()` casts the event to the appropriate class-specific structure and then checks its subclass to determine which subclass counter to increment. For example, the code to handle events in the connection-event class looks like this:

```
else if (event_class == MYSQL_AUDIT_CONNECTION_CLASS)
{
  const struct mysql_event_connection *event_connection=
                                (const struct mysql_event_connection *) event;

  switch (event_connection->event_subclass)
  {
  case MYSQL_AUDIT_CONNECTION_CONNECT:
    number_of_calls_connection_connect++;
    break;
  case MYSQL_AUDIT_CONNECTION_DISCONNECT:
    number_of_calls_connection_disconnect++;
    break;
  case MYSQL_AUDIT_CONNECTION_CHANGE_USER:
    number_of_calls_connection_change_user++;
```

```
    break;
  case MYSQL_AUDIT_CONNECTION_PRE_AUTHENTICATE:
    number_of_calls_connection_pre_authenticate++;
      break;
  default:
    break;
  }
}
```

> **Note**
>
> The general event class (`MYSQL_AUDIT_GENERAL_CLASS`) is deprecated and will be removed in a future MySQL release. To reduce plugin overhead, it is preferable to subscribe only to the more specific event classes of interest.

For some event classes, the `NULL_AUDIT` plugin performs other processing in addition to incrementing a counter. In any case, when the notification function finishes processing the event, it should return a status indicating whether the server should continue processing the event or terminate it.

Audit Plugin Error Handling

Audit plugin notification functions can report a status value for the current event two ways:

- Use the notification function return value. In this case, the function returns zero if the server should continue processing the event, or nonzero if the server should terminate the event.
- Call the `my_message()` function to set the error state before returning from the notification function. In this case, the notification function return value is ignored and the server aborts the event and terminates event processing with an error. The `my_message()` arguments indicate which error to report, and its message. For example:

  ```
  my_message(ER_AUDIT_API_ABORT, "This is my error message.", MYF(0));
  ```

  Some events cannot be aborted. A nonzero return value is not taken into consideration and the `my_message()` error call must follow an `is_error()` check. For example:

  ```
  if (!thd->get_stmt_da()->is_error())
  {
    my_message(ER_AUDIT_API_ABORT, "This is my error message.", MYF(0));
  }
  ```

These events cannot be aborted:

- `MYSQL_AUDIT_CONNECTION_DISCONNECT`: The server cannot prevent a client from disconnecting.
- `MYSQL_AUDIT_COMMAND_END`: This event provides the status of a command that has finished executing, so there is no purpose to terminating it.

If an audit plugin returns nonzero status for a nonterminable event, the server ignores the status and continues processing the event. This is also true if an audit plugin uses the `my_message()` function to terminate a nonterminable event.

Audit Plugin Usage

To compile and install a plugin library file, use the instructions in Section 4.4.3, “Compiling and Installing Plugin Libraries”. To make the library file available for use, install it in the plugin directory (the directory named by the `plugin_dir` system variable). For the `NULL_AUDIT` plugin, it is compiled and installed when you build MySQL from source. It is also included in binary distributions. The build process produces

a shared object library with a name of `adt_null.so` (the `.so` suffix might differ depending on your platform).

To register the plugin at runtime, use this statement, adjusting the `.so` suffix for your platform as necessary:

```
INSTALL PLUGIN NULL_AUDIT SONAME 'adt_null.so';
```

For additional information about plugin loading, see Installing and Uninstalling Plugins.

To verify plugin installation, examine the `INFORMATION_SCHEMA.PLUGINS` table or use the `SHOW PLUGINS` statement. See Obtaining Server Plugin Information.

While the `NULL_AUDIT` audit plugin is installed, it exposes status variables that indicate the events for which the plugin has been called:

```
mysql> SHOW STATUS LIKE 'Audit_null%';
+----------------------------------------+--------+
| Variable_name                          | Value  |
+----------------------------------------+--------+
| Audit_null_authorization_column        | 0      |
| Audit_null_authorization_db            | 0      |
| Audit_null_authorization_procedure     | 0      |
| Audit_null_authorization_proxy         | 0      |
| Audit_null_authorization_table         | 0      |
| Audit_null_authorization_user          | 0      |
| Audit_null_called                      | 185547 |
| Audit_null_command_end                 | 20999  |
| Audit_null_command_start               | 21001  |
| Audit_null_connection_change_user      | 0      |
| Audit_null_connection_connect          | 5823   |
| Audit_null_connection_disconnect       | 5818   |
| Audit_null_connection_pre_authenticate | 5823   |
| Audit_null_general_error               | 1      |
| Audit_null_general_log                 | 26559  |
| Audit_null_general_result              | 19922  |
| Audit_null_general_status              | 21000  |
| Audit_null_global_variable_get         | 0      |
| Audit_null_global_variable_set         | 0      |
| Audit_null_parse_postparse             | 14648  |
| Audit_null_parse_preparse              | 14648  |
| Audit_null_query_nested_start          | 6      |
| Audit_null_query_nested_status_end     | 6      |
| Audit_null_query_start                 | 14648  |
| Audit_null_query_status_end            | 14647  |
| Audit_null_server_shutdown             | 0      |
| Audit_null_server_startup              | 1      |
| Audit_null_table_access_delete         | 104    |
| Audit_null_table_access_insert         | 2839   |
| Audit_null_table_access_read           | 97842  |
| Audit_null_table_access_update         | 278    |
+----------------------------------------+--------+
```

`Audit_null_called` counts all events, and the other variables count instances of specific event subclasses. For example, the preceding `SHOW STATUS` statement causes the server to send a result to the client and to write a message to the general query log if that log is enabled. Thus, a client that issues the statement repeatedly causes `Audit_null_called`, `Audit_null_general_result`, and `Audit_null_general_log` to be incremented each time.

The status variables values are global and aggregated across all sessions. There are no counters for individual sessions.

`NULL_AUDIT` exposes several system variables that enable communication with the plugin at runtime:

```
mysql> SHOW VARIABLES LIKE 'null_audit%';
+-----------------------------------------------+-------+
| Variable_name                                 | Value |
+-----------------------------------------------+-------+
| null_audit_abort_message                      |       |
| null_audit_abort_value                        | 1     |
| null_audit_event_order_check                  |       |
| null_audit_event_order_check_consume_ignore_count | 0     |
| null_audit_event_order_check_exact            | 1     |
| null_audit_event_order_started                | 0     |
| null_audit_event_record                       |       |
| null_audit_event_record_def                   |       |
+-----------------------------------------------+-------+
```

The `NULL_AUDIT` system variables have these meanings:

- `null_audit_abort_message`: The custom error message to use when an event is aborted.
- `null_audit_abort_value`: The custom error code to use when an event is aborted.
- `null_audit_event_order_check`: Prior to event matching, the expected event order. After event matching, the matching outcome.
- `null_audit_event_order_check_consume_ignore_count`: Number of times event matching should not consume matched events.
- `null_audit_event_order_check_exact`: Whether event matching must be exact. Disabling this variable enables skipping events not listed in `null_audit_event_order_check` during event-order matching. Of the events specified, they must still match in the order given.
- `null_audit_event_order_started`: For internal use.
- `null_audit_event_record`: The recorded events after event recording takes place.
- `null_audit_event_record_def`: The names of the start and end events to match when recording events, separated by a semicolon. The value must be set before each statement for which events are recorded.

To demonstrate use of those system variables, suppose that a table `db1.t1` exists, created as follows:

```
CREATE DATABASE db1;
CREATE TABLE db1.t1 (a VARCHAR(255));
```

For test-creation purposes, it is possible to record events that pass through the plugin. To start recording, specify the start and end events in the `null_audit_event_record_def` variable. For example:

```
SET @@null_audit_event_record_def =
  'MYSQL_AUDIT_COMMAND_START;MYSQL_AUDIT_COMMAND_END';
```

After a statement occurs that matches those start and end events, the `null_audit_event_record` system variable contains the resulting event sequence. For example, after recording the events for a `SELECT 1` statement, `null_audit_event_record` is a string that has a value consisting of a set of event strings:

```
MYSQL_AUDIT_COMMAND_START;command_id="3";
MYSQL_AUDIT_PARSE_PREPARSE;;
MYSQL_AUDIT_PARSE_POSTPARSE;;
MYSQL_AUDIT_GENERAL_LOG;;
MYSQL_AUDIT_QUERY_START;sql_command_id="0";
MYSQL_AUDIT_QUERY_STATUS_END;sql_command_id="0";
MYSQL_AUDIT_GENERAL_RESULT;;
```

```
MYSQL_AUDIT_GENERAL_STATUS;;
MYSQL_AUDIT_COMMAND_END;command_id="3";
```

After recording the events for an `INSERT INTO db1.t1 VALUES ('some data')` statement, `null_audit_event_record` has this value:

```
MYSQL_AUDIT_COMMAND_START;command_id="3";
MYSQL_AUDIT_PARSE_PREPARSE;;
MYSQL_AUDIT_PARSE_POSTPARSE;;
MYSQL_AUDIT_GENERAL_LOG;;
MYSQL_AUDIT_QUERY_START;sql_command_id="5";
MYSQL_AUDIT_TABLE_ACCESS_INSERT;db="db1" table="t1";
MYSQL_AUDIT_QUERY_STATUS_END;sql_command_id="5";
MYSQL_AUDIT_GENERAL_RESULT;;
MYSQL_AUDIT_GENERAL_STATUS;;
MYSQL_AUDIT_COMMAND_END;command_id="3";
```

Each event string has this format, with semicolons separating the string parts:

```
event_name;event_data;command
```

Event strings have these parts:

- *`event_name`*: The event name (a symbol that begins with `MYSQL_AUDIT_`).
- *`event_data`*: Empty, or, as described later, data associated with the event.
- *`command`*: Empty, or, as described later, a command to execute when the event is matched.

> **Note**
>
> A limitation of the `NULL_AUDIT` plugin is that event recording works for a single session only. Once you record events in a given session, event recording in subsequent sessions yields a `null_audit_event_record` value of `NULL`. To record events again, it is necessary to restart the plugin.

To check the order of audit API calls, set the `null_audit_event_order_check` variable to the expected event order for a particular operation, listing one or more event strings, each containing two semicolons internally, with additional semicolons separating adjacent event strings:

```
event_name;event_data;command [;event_name;event_data;command] ...
```

For example:

```
SET @@null_audit_event_order_check =
  'MYSQL_AUDIT_CONNECTION_PRE_AUTHENTICATE;;;'
  'MYSQL_AUDIT_GENERAL_LOG;;;'
  'MYSQL_AUDIT_CONNECTION_CONNECT;;';
```

For better readability, the statement takes advantage of the SQL syntax that concatenates adjacent strings into a single string.

After you set the `null_audit_event_order_check` variable to a list of event strings, the next matching operation replaces the variable value with a value that indicates the operation outcome:

- If the expected event order was matched successfully, the resulting `null_audit_event_order_check` value is `EVENT-ORDER-OK`.
- If the `null_audit_event_order_check` value specified aborting a matched event (as described later), the resulting `null_audit_event_order_check` value is `EVENT-ORDER-ABORT`.

- If the expected event order failed with unexpected data, the resulting `null_audit_event_order_check` value is `EVENT-ORDER-INVALID-DATA`. This occurs, for example, if an event was specified as expected to affect table `t1` but actually affected `t2`.

When you assign to `null_audit_event_order_check` the list of events to be matched, some events should be specified with a nonempty *`event_data`* part of the event string. The following table shows the *`event_data`* format for these events. If an event takes multiple data values, they must be specified in the order shown. Alternatively, it is possible to specify an *`event_data`* value as `<IGNORE>` to ignore event data content; in this case, it does not matter whether or not an event haas data.

Applicable Events	Event Data Format

`MYSQL_AUDIT_COMMAND_START`
 	
`MYSQL_AUDIT_COMMAND_END`
	`command_id="id_value"`

`MYSQL_AUDIT_GLOBAL_VARIABLE_GET`
 	
`MYSQL_AUDIT_GLOBAL_VARIABLE_SET`
	`name="var_value" value="var_value"`

`MYSQL_AUDIT_QUERY_NESTED_START`
 	
`MYSQL_AUDIT_QUERY_NESTED_STATUS_END`
 	
`MYSQL_AUDIT_QUERY_START`
 	
`MYSQL_AUDIT_QUERY_STATUS_END`
	`sql_command_id="id_value"`

`MYSQL_AUDIT_TABLE_ACCESS_DELETE`
 	
`MYSQL_AUDIT_TABLE_ACCESS_INSERT`
 	
`MYSQL_AUDIT_TABLE_ACCESS_READ`
 	
`MYSQL_AUDIT_TABLE_ACCESS_UPDATE`
	`db="db_name" table="table_name"`

In the `null_audit_event_order_check` value, specifying `ABORT_RET` in the *`command`* part of an event string makes it possible to abort the audit API call on the specified event. (Assuming that the event is one that can be aborted. Those that cannot were described previously.) For example, as shown previously, this is the expected order of events for an insert into `t1`:

```
MYSQL_AUDIT_COMMAND_START;command_id="3";
MYSQL_AUDIT_PARSE_PREPARSE;;
MYSQL_AUDIT_PARSE_POSTPARSE;;
MYSQL_AUDIT_GENERAL_LOG;;
MYSQL_AUDIT_QUERY_START;sql_command_id="5";
MYSQL_AUDIT_TABLE_ACCESS_INSERT;db="db1" table="t1";
MYSQL_AUDIT_QUERY_STATUS_END;sql_command_id="5";
MYSQL_AUDIT_GENERAL_RESULT;;
MYSQL_AUDIT_GENERAL_STATUS;;
MYSQL_AUDIT_COMMAND_END;command_id="3";
```

To abort `INSERT` statement execution when the `MYSQL_AUDIT_QUERY_STATUS_END` event occurs, set `null_audit_event_order_check` like this (remember to add semicolon separators between adjacent event strings):

```
SET @@null_audit_event_order_check =
  'MYSQL_AUDIT_COMMAND_START;command_id="3";;'
  'MYSQL_AUDIT_PARSE_PREPARSE;;;'
  'MYSQL_AUDIT_PARSE_POSTPARSE;;;'
  'MYSQL_AUDIT_GENERAL_LOG;;;'
```

```
  'MYSQL_AUDIT_QUERY_START;sql_command_id="5";;'
  'MYSQL_AUDIT_TABLE_ACCESS_INSERT;db="db1" table="t1";;'
  'MYSQL_AUDIT_QUERY_STATUS_END;sql_command_id="5";ABORT_RET';
```

It is not necesary to list events that are expected to occur after the event string that contains a *`command`* value of `ABORT_RET`.

After the audit plugin matches the preceding sequence, it aborts event processing and sends an error message to the client. It also sets `null_audit_event_order_check` to `EVENT-ORDER-ABORT`:

```
mysql> INSERT INTO db1.t1 VALUES ('some data');
ERROR 3164 (HY000): Aborted by Audit API ('MYSQL_AUDIT_QUERY_STATUS_END';1).
mysql> SELECT @@null_audit_event_order_check;
+-------------------------------+
| @@null_audit_event_order_check |
+-------------------------------+
| EVENT-ORDER-ABORT              |
+-------------------------------+
```

Returning a nonzero value from the audit API notification routine is the standard way to abort event execution. It is also possible to specify a custom error code by setting the `null_audit_abort_value` variable to the value that the notification routine should return:

```
SET @@null_audit_abort_value = 123;
```

Aborting a sequence results in a standard message with the custom error code. Suppose that you set audit log system variables like this, to abort on a match for the events that occur for a `SELECT 1` statement:

```
SET @@null_audit_abort_value = 123;
SET @@null_audit_event_order_check =
  'MYSQL_AUDIT_COMMAND_START;command_id="3";;'
  'MYSQL_AUDIT_PARSE_PREPARSE;;;'
  'MYSQL_AUDIT_PARSE_POSTPARSE;;;'
  'MYSQL_AUDIT_GENERAL_LOG;;;'
  'MYSQL_AUDIT_QUERY_START;sql_command_id="0";ABORT_RET';
```

Then execution of `SELECT 1` results in this error message that includes the custom error code:

```
mysql> SELECT 1;
ERROR 3164 (HY000): Aborted by Audit API ('MYSQL_AUDIT_QUERY_START';123).

mysql> SELECT @@null_audit_event_order_check;
+-------------------------------+
| @@null_audit_event_order_check |
+-------------------------------+
| EVENT-ORDER-ABORT              |
+-------------------------------+
```

An event can be also aborted with a custom message, specified by setting the `null_audit_abort_message` variable. Suppose that you set audit log system variables like this:

```
SET @@null_audit_abort_message = 'Custom error text.';
SET @@null_audit_event_order_check =
  'MYSQL_AUDIT_COMMAND_START;command_id="3";;'
  'MYSQL_AUDIT_PARSE_PREPARSE;;;'
  'MYSQL_AUDIT_PARSE_POSTPARSE;;;'
  'MYSQL_AUDIT_GENERAL_LOG;;;'
  'MYSQL_AUDIT_QUERY_START;sql_command_id="0";ABORT_RET';
```

Then aborting a sequence results in the following error message:

```
mysql> SELECT 1;
ERROR 3164 (HY000): Custom error text.
```

```
mysql> SELECT @@null_audit_event_order_check;
+--------------------------------+
| @@null_audit_event_order_check |
+--------------------------------+
| EVENT-ORDER-ABORT              |
+--------------------------------+
```

To disable the `NULL_AUDIT` plugin after testing it, use this statement to unload it:

```
UNINSTALL PLUGIN NULL_AUDIT;
```

4.4.9 Writing Authentication Plugins

MySQL supports pluggable authentication, in which plugins are invoked to authenticate client connections. Authentication plugins enable the use of authentication methods other than the built-in method of passwords stored in the `mysql.user` system table. For example, plugins can be written to access external authentication methods. Also, authentication plugins can support the proxy user capability, such that the connecting user is a proxy for another user and is treated, for purposes of access control, as having the privileges of a different user. For more information, see Pluggable Authentication, and Proxy Users.

An authentication plugin can be written for the server side or the client side. Server-side plugins use the same plugin API that is used for the other server plugin types such as full-text parser or audit plugins (although with a different type-specific descriptor). Client-side plugins use the client plugin API.

Several header files contain information relevant to authentication plugins:

- `plugin.h`: Defines the `MYSQL_AUTHENTICATION_PLUGIN` server plugin type.
- `client_plugin.h`: Defines the API for client plugins. This includes the client plugin descriptor and function prototypes for client plugin C API calls (see C API Client Plugin Interface).
- `plugin_auth.h`: Defines the part of the server plugin API specific to authentication plugins. This includes the type-specific descriptor for server-side authentication plugins and the `MYSQL_SERVER_AUTH_INFO` structure.
- `plugin_auth_common.h`: Contains common elements of client and server authentication plugins. This includes return value definitions and the `MYSQL_PLUGIN_VIO` structure.

To write an authentication plugin, include the following header files in the plugin source file. Other MySQL or general header files might also be needed, depending on the plugin capabilities and requirements.

- For a source file that implements a server authentication plugin, include this file:

  ```
  #include <mysql/plugin_auth.h>
  ```

- For a source file that implements a client authentication plugin, or both client and server plugins, include these files:

  ```
  #include <mysql/plugin_auth.h>
  #include <mysql/client_plugin.h>
  #include <mysql.h>
  ```

`plugin_auth.h` includes `plugin.h` and `plugin_auth_common.h`, so you need not include the latter files explicitly.

This section describes how to write a pair of simple server and client authentication plugins that work together.

> **Warning**
>
> These plugins accept any non-empty password and the password is sent as cleartext. This is insecure, so the plugins *should not be used in production environments.*

The server-side and client-side plugins developed here both are named `auth_simple`. As described in Section 4.4.2, “Plugin Data Structures”, the plugin library file must have the same base name as the client plugin, so the source file name is `auth_simple.c` and produces a library named `auth_simple.so` (assuming that your system uses `.so` as the suffix for library files).

In MySQL source distributions, authentication plugin source is located in the `plugin/auth` directory and can be examined as a guide to writing other authentication plugins. Also, to see how the built-in authentication plugins are implemented, see `sql/sql_acl.cc` for plugins that are built in to the MySQL server and `sql-common/client.c` for plugins that are built in to the `libmysqlclient` client library. (For the built-in client plugins, note that the `auth_plugin_t` structures used there differ from the structures used with the usual client plugin declaration macros. In particular, the first two members are provided explicitly, not by declaration macros.)

4.4.9.1 Writing the Server-Side Authentication Plugin

Declare the server-side plugin with the usual general descriptor format that is used for all server plugin types (see Section 4.4.2.1, “Server Plugin Library and Plugin Descriptors”). For the `auth_simple` plugin, the descriptor looks like this:

```
mysql_declare_plugin(auth_simple)
{
  MYSQL_AUTHENTICATION_PLUGIN,
  &auth_simple_handler,                /* type-specific descriptor */
  "auth_simple",                       /* plugin name */
  "Author Name",                       /* author */
  "Any-password authentication plugin", /* description */
  PLUGIN_LICENSE_GPL,                  /* license type */
  NULL,                                /* no init function */
  NULL,                                /* no deinit function */
  0x0100,                              /* version = 1.0 */
  NULL,                                /* no status variables */
  NULL,                                /* no system variables */
  NULL,                                /* no reserved information */
  0                                    /* no flags */
}
mysql_declare_plugin_end;
```

The `name` member (`auth_simple`) indicates the name to use for references to the plugin in statements such as `INSTALL PLUGIN` or `UNINSTALL PLUGIN`. This is also the name displayed by `SHOW PLUGINS` or `INFORMATION_SCHEMA.PLUGINS`.

The `auth_simple_handler` member of the general descriptor points to the type-specific descriptor. For an authentication plugin, the type-specific descriptor is an instance of the `st_mysql_auth` structure (defined in `plugin_auth.h`):

```
struct st_mysql_auth
{
  int interface_version;
  const char *client_auth_plugin;
  int (*authenticate_user)(MYSQL_PLUGIN_VIO *vio, MYSQL_SERVER_AUTH_INFO *info);
  int (*generate_authentication_string)(char *outbuf,
      unsigned int *outbuflen, const char *inbuf, unsigned int inbuflen);
  int (*validate_authentication_string)(char* const inbuf, unsigned int buflen);
  int (*set_salt)(const char *password, unsigned int password_len,
                  unsigned char* salt, unsigned char *salt_len);
```

```
  const unsigned long authentication_flags;
};
```

The `st_mysql_auth` structure has these members:

- `interface_version`: The type-specific API version number, always `MYSQL_AUTHENTICATION_INTERFACE_VERSION`
- `client_auth_plugin`: The client plugin name
- `authenticate_user`: A pointer to the main plugin function that communicates with the client
- `generate_authentication_string`: A pointer to a plugin function that generates a password digest from an authentication string
- `validate_authentication_string`: A pointer to a plugin function that validates a password digest
- `set_salt`: A pointer to a plugin function that converts a scrambled password to binary form
- `authentication_flags`: A flags word

The `client_auth_plugin` member should indicate the name of the client plugin if a specific plugin is required. A value of `NULL` means “any plugin.” In the latter case, whatever plugin the client uses will do. This is useful if the server plugin does not care about the client plugin or what user name or password it sends. For example, this might be true if the server plugin authenticates only local clients and uses some property of the operating system rather than the information sent by the client plugin.

For `auth_simple`, the type-specific descriptor looks like this:

```
static struct st_mysql_auth auth_simple_handler =
{
  MYSQL_AUTHENTICATION_INTERFACE_VERSION,
  "auth_simple",             /* required client-side plugin name */
  auth_simple_server         /* server-side plugin main function */
  generate_auth_string_hash, /* generate digest from password string */
  validate_auth_string_hash, /* validate password digest */
  set_salt,                  /* generate password salt value */
  AUTH_FLAG_PRIVILEGED_USER_FOR_PASSWORD_CHANGE
};
```

The main function, `auth_simple_server()`, takes two arguments representing an I/O structure and a `MYSQL_SERVER_AUTH_INFO` structure. The structure definition, found in `plugin_auth.h`, looks like this:

```
typedef struct st_mysql_server_auth_info
{
  char *user_name;
  unsigned int user_name_length;
  const char *auth_string;
  unsigned long auth_string_length;
  char authenticated_as[MYSQL_USERNAME_LENGTH+1];
  char external_user[512];
  int  password_used;
  const char *host_or_ip;
  unsigned int host_or_ip_length;
} MYSQL_SERVER_AUTH_INFO;
```

The character set for string members is UTF-8. If there is a `_length` member associated with a string, it indicates the string length in bytes. Strings are also null-terminated.

When an authentication plugin is invoked by the server, it should interpret the `MYSQL_SERVER_AUTH_INFO` structure members as follows. Some of these are used to set the value of SQL functions or system variables within the client session, as indicated.

- `user_name`: The user name sent by the client. The value becomes the `USER()` function value.
- `user_name_length`: The length of `user_name` in bytes.
- `auth_string`: The value of the `authentication_string` column of the row in the `mysql.user` system table for the matching account name (that is, the row that matches the client user name and host name and that the server uses to determine how to authenticate the client).

  Suppose that you create an account using the following statement:

  ```
  CREATE USER 'my_user'@'localhost'
    IDENTIFIED WITH my_plugin AS 'my_auth_string';
  ```

  When `my_user` connects from the local host, the server invokes `my_plugin` and passes `'my_auth_string'` to it as the `auth_string` value.
- `auth_string_length`: The length of `auth_string` in bytes.
- `authenticated_as`: The server sets this to the user name (the value of `user_name`). The plugin can alter it to indicate that the client should have the privileges of a different user. For example, if the plugin supports proxy users, the initial value is the name of the connecting (proxy) user, and the plugin can change this member to the proxied user name. The server then treats the proxy user as having the privileges of the proxied user (assuming that the other conditions for proxy user support are satisfied; see Section 4.4.9.4, "Implementing Proxy User Support in Authentication Plugins"). The value is represented as a string at most `MYSQL_USER_NAME_LENGTH` bytes long, plus a terminating null. The value becomes the `CURRENT_USER()` function value.
- `external_user`: The server sets this to the empty string (null terminated). Its value becomes the `external_user` system variable value. If the plugin wants that system variable to have a different value, it should set this member accordingly (for example, to the connecting user name). The value is represented as a string at most 511 bytes long, plus a terminating null.
- `password_used`: This member applies when authentication fails. The plugin can set it or ignore it. The value is used to construct the failure error message of `Authentication fails. Password used: %s`. The value of `password_used` determines how `%s` is handled, as shown in the following table.

`password_used`	`%s` Handling
0	NO
1	YES
2	There will be no `%s`

- `host_or_ip`: The name of the client host if it can be resolved, or the IP address otherwise.
- `host_or_ip_length`: The length of `host_or_ip` in bytes.

The `auth_simple` main function, `auth_simple_server()`, reads the password (a null-terminated string) from the client and succeeds if the password is nonempty (first byte not null):

```
static int auth_simple_server (MYSQL_PLUGIN_VIO *vio,
                               MYSQL_SERVER_AUTH_INFO *info)
{
  unsigned char *pkt;
  int pkt_len;

  /* read the password as null-terminated string, fail on error */
  if ((pkt_len= vio->read_packet(vio, &pkt)) < 0)
    return CR_ERROR;
```

```
  /* fail on empty password */
  if (!pkt_len || *pkt == '\0')
  {
    info->password_used= PASSWORD_USED_NO;
    return CR_ERROR;
  }

  /* accept any nonempty password */
  info->password_used= PASSWORD_USED_YES;

  return CR_OK;
}
```

The main function should return one of the error codes shown in the following table.

Error Code	Meaning
`CR_OK`	Success
`CR_OK_HANDSHAKE_COMPLETE`	Do not send a status packet back to client
`CR_ERROR`	Error
`CR_AUTH_USER_CREDENTIALS`	Authentication failure
`CR_AUTH_HANDSHAKE`	Authentication handshake failure
`CR_AUTH_PLUGIN_ERROR`	Internal plugin error

For an example of how the handshake works, see the `plugin/auth/dialog.c` source file.

The server counts plugin errors in the Performance Schema `host_cache` table.

`auth_simple_server()` is so basic that it does not use the authentication information structure except to set the member that indicates whether a password was received.

A plugin that supports proxy users must return to the server the name of the proxied user (the MySQL user whose privileges the client user should get). To do this, the plugin must set the `info->authenticated_as` member to the proxied user name. For information about proxying, see Proxy Users, and Section 4.4.9.4, “Implementing Proxy User Support in Authentication Plugins”.

The `generate_authentication_string` member of the plugin descriptor takes the password and generates a password hash (digest) from it:

- The first two arguments are pointers to the output buffer and its maximum length in bytes. The function should write the password hash to the output buffer and reset the length to the actual hash length.
- The second two arguments indicate the password input buffer and its length in bytes.
- The function returns 0 for success, 1 if an error occurred.

For the `auth_simple` plugin, the `generate_auth_string_hash()` function implements the `generate_authentication_string` member. It just makes a copy of the password, unless it is too long to fit in the output buffer.

```
int generate_auth_string_hash(char *outbuf, unsigned int *buflen,
                              const char *inbuf, unsigned int inbuflen)
{
  /*
    fail if buffer specified by server cannot be copied to output buffer
  */
  if (*buflen < inbuflen)
    return 1;   /* error */
```

```
  strncpy(outbuf, inbuf, inbuflen);
  *buflen= strlen(inbuf);
  return 0;     /* success */
}
```

The `validate_authentication_string` member of the plugin descriptor validates a password hash:

- The arguments are a pointer to the password hash and its length in bytes.
- The function returns 0 for success, 1 if the password hash cannot be validated.

For the `auth_simple` plugin, the `validate_auth_string_hash()` function implements the `validate_authentication_string` member. It returns success unconditionally:

```
int validate_auth_string_hash(char* const inbuf  __attribute__((unused)),
                              unsigned int buflen  __attribute__((unused)))
{
  return 0;     /* success */
}
```

The `set_salt` member of the plugin descriptor is used only by the `mysql_native_password` plugin (see Native Pluggable Authentication). For other authentication plugins, you can use this trivial implementation:

```
int set_salt(const char* password __attribute__((unused)),
             unsigned int password_len __attribute__((unused)),
             unsigned char* salt __attribute__((unused)),
             unsigned char* salt_len)
{
  *salt_len= 0;
  return 0;     /* success */
}
```

The `authentication_flags` member of the plugin descriptor contains flags that affect plugin operation. The permitted flags are:

- `AUTH_FLAG_PRIVILEGED_USER_FOR_PASSWORD_CHANGE`: Credential changes are a privileged operation. If this flag is set, the server requires that the user has the global `CREATE USER` privilege or the `UPDATE` privilege for the `mysql` database.
- `AUTH_FLAG_USES_INTERNAL_STORAGE`: Whether the plugin uses internal storage (in the `authentication_string` column of `mysql.user` rows). If this flag is not set, attempts to set the password fail and the server produces a warning.

4.4.9.2 Writing the Client-Side Authentication Plugin

Declare the client-side plugin descriptor with the `mysql_declare_client_plugin()` and `mysql_end_client_plugin` macros (see Section 4.4.2.3, “Client Plugin Descriptors”). For the `auth_simple` plugin, the descriptor looks like this:

```
mysql_declare_client_plugin(AUTHENTICATION)
  "auth_simple",                        /* plugin name */
  "Author Name",                        /* author */
  "Any-password authentication plugin", /* description */
  {1,0,0},                              /* version = 1.0.0 */
  "GPL",                                /* license type */
  NULL,                                 /* for internal use */
  NULL,                                 /* no init function */
  NULL,                                 /* no deinit function */
  NULL,                                 /* no option-handling function */
  auth_simple_client                    /* main function */
```

```
mysql_end_client_plugin;
```

The descriptor members from the plugin name through the option-handling function are common to all client plugin types. (For descriptions, see Section 4.4.2.3, “Client Plugin Descriptors”.) Following the common members, the descriptor has an additional member specific to authentication plugins. This is the “main” function, which handles communication with the server. The function takes two arguments representing an I/O structure and a connection handler. For our simple any-password plugin, the main function does nothing but write to the server the password provided by the user:

```
static int auth_simple_client (MYSQL_PLUGIN_VIO *vio, MYSQL *mysql)
{
  int res;

  /* send password as null-terminated string as cleartext */
  res= vio->write_packet(vio, (const unsigned char *) mysql->passwd,
                         strlen(mysql->passwd) + 1);

  return res ? CR_ERROR : CR_OK;
}
```

The main function should return one of the error codes shown in the following table.

Error Code	Meaning
`CR_OK`	Success
`CR_OK_HANDSHAKE_COMPLETE`	Success, client done
`CR_ERROR`	Error

`CR_OK_HANDSHAKE_COMPLETE` indicates that the client has done its part successfully and has read the last packet. A client plugin may return `CR_OK_HANDSHAKE_COMPLETE` if the number of round trips in the authentication protocol is not known in advance and the plugin must read another packet to determine whether authentication is finished.

4.4.9.3 Using the Authentication Plugins

To compile and install a plugin library file, use the instructions in Section 4.4.3, “Compiling and Installing Plugin Libraries”. To make the library file available for use, install it in the plugin directory (the directory named by the `plugin_dir` system variable).

Register the server-side plugin with the server. For example, to load the plugin at server startup, use a `--plugin-load=auth_simple.so` option, adjusting the `.so` suffix for your platform as necessary.

Create a user for whom the server will use the `auth_simple` plugin for authentication:

```
mysql> CREATE USER 'x'@'localhost'
    -> IDENTIFIED WITH auth_simple;
```

Use a client program to connect to the server as user `x`. The server-side `auth_simple` plugin communicates with the client program that it should use the client-side `auth_simple` plugin, and the latter sends the password to the server. The server plugin should reject connections that send an empty password and accept connections that send a nonempty password. Invoke the client program each way to verify this:

```
shell> mysql --user=x --skip-password
ERROR 1045 (28000): Access denied for user 'x'@'localhost' (using password: NO)

shell> mysql --user=x --password
Enter password: abc
mysql>
```

Because the server plugin accepts any nonempty password, it should be considered insecure. After testing the plugin to verify that it works, restart the server without the `--plugin-load` option so as not to indavertently leave the server running with an insecure authentication plugin loaded. Also, drop the user with `DROP USER 'x'@'localhost'`.

For additional information about loading and using authentication plugins, see Installing and Uninstalling Plugins, and Pluggable Authentication.

If you are writing a client program that supports the use of authentication plugins, normally such a program 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.

Should a client program require lower-level plugin management, the client library contains functions that take an `st_mysql_client_plugin` argument. See C API Client Plugin Interface.

4.4.9.4 Implementing Proxy User Support in Authentication Plugins

One of the capabilities that pluggable authentication makes possible is proxy users (see Proxy Users). For a server-side authentication plugin to participate in proxy user support, these conditions must be satisfied:

- When a connecting client should be treated as a proxy user, the plugin must return a different name in the `authenticated_as` member of the `MYSQL_SERVER_AUTH_INFO` structure, to indicate the proxied user name. It may also optionally set the `external_user` member, to set the value of the `external_user` system variable.
- Proxy user accounts must be set up to be authenticated by the plugin. Use the `CREATE USER` or `GRANT` statement to associate accounts with plugins.
- Proxy user accounts must have the `PROXY` privilege for the proxied accounts. Use the `GRANT` statement to grant this privilege.

In other words, the only aspect of proxy user support required of the plugin is that it set `authenticated_as` to the proxied user name. The rest is optional (setting `external_user`) or done by the DBA using SQL statements.

How does an authentication plugin determine which proxied user to return when the proxy user connects? That depends on the plugin. Typically, the plugin maps clients to proxied users based on the authentication string passed to it by the server. This string comes from the `AS` part of the `IDENTIFIED WITH` clause of the `CREATE USER` statement that specifies use of the plugin for authentication.

The plugin developer determines the syntax rules for the authentication string and implements the plugin according to those rules. Suppose that a plugin takes a comma-separated list of pairs that map external users to MySQL users. For example:

```
CREATE USER ''@'%.example.com'
  IDENTIFIED WITH my_plugin AS 'extuser1=mysqlusera, extuser2=mysqluserb'
CREATE USER ''@'%.example.org'
```

```
  IDENTIFIED WITH my_plugin AS 'extuser1=mysqluserc, extuser2=mysqluserd'
```

When the server invokes a plugin to authenticate a client, it passes the appropriate authentication string to the plugin. The plugin is responsible to:

1. Parse the string into its components to determine the mapping to use
2. Compare the client user name to the mapping
3. Return the proper MySQL user name

For example, if `extuser2` connects from an `example.com` host, the server passes `'extuser1=mysqlusera, extuser2=mysqluserb'` to the plugin, and the plugin should copy `mysqluserb` into `authenticated_as`, with a terminating null byte. If `extuser2` connects from an `example.org` host, the server passes `'extuser1=mysqluserc, extuser2=mysqluserd'`, and the plugin should copy `mysqluserd` instead.

If there is no match in the mapping, the action depends on the plugin. If a match is required, the plugin likely will return an error. Or the plugin might simply return the client name; in this case, it should not change `authenticated_as`, and the server will not treat the client as a proxy.

The following example demonstrates how to handle proxy users using a plugin named `auth_simple_proxy`. Like the `auth_simple` plugin described earlier, `auth_simple_proxy` accepts any nonempty password as valid (and thus should not be used in production environments). In addition, it examines the `auth_string` authentication string member and uses these very simple rules for interpreting it:

- If the string is empty, the plugin returns the user name as given and no proxying occurs. That is, the plugin leaves the value of `authenticated_as` unchanged.
- If the string is nonempty, the plugin treats it as the name of the proxied user and copies it to `authenticated_as` so that proxying occurs.

For testing, set up one account that is not proxied according to the preceding rules, and one that is. This means that one account has no `AS` clause, and one includes an `AS` clause that names the proxied user:

```
CREATE USER 'plugin_user1'@'localhost'
  IDENTIFIED WITH auth_simple_proxy;
CREATE USER 'plugin_user2'@'localhost'
  IDENTIFIED WITH auth_simple_proxy AS 'proxied_user';
```

In addition, create an account for the proxied user and grant `plugin_user2` the `PROXY` privilege for it:

```
CREATE USER 'proxied_user'@'localhost'
  IDENTIFIED BY 'proxied_user_pass';
GRANT PROXY
  ON 'proxied_user'@'localhost'
  TO 'plugin_user2'@'localhost';
```

Before the server invokes an authentication plugin, it sets `authenticated_as` to the client user name. To indicate that the user is a proxy, the plugin should set `authenticated_as` to the proxied user name. For `auth_simple_proxy`, this means that it must examine the `auth_string` value, and, if the value is nonempty, copy it to the `authenticated_as` member to return it as the name of the proxied user. In addition, when proxying occurs, the plugin sets the `external_user` member to the client user name; this becomes the value of the `external_user` system variable.

```
static int auth_simple_proxy_server (MYSQL_PLUGIN_VIO *vio,
                                     MYSQL_SERVER_AUTH_INFO *info)
{
  unsigned char *pkt;
```

```
  int pkt_len;

  /* read the password as null-terminated string, fail on error */
  if ((pkt_len= vio->read_packet(vio, &pkt)) < 0)
    return CR_ERROR;

  /* fail on empty password */
  if (!pkt_len || *pkt == '\0')
  {
    info->password_used= PASSWORD_USED_NO;
    return CR_ERROR;
  }

  /* accept any nonempty password */
  info->password_used= PASSWORD_USED_YES;

  /* if authentication string is nonempty, use as proxied user name */
  /* and use client name as external_user value */
  if (info->auth_string_length > 0)
  {
    strcpy (info->authenticated_as, info->auth_string);
    strcpy (info->external_user, info->user_name);
  }

  return CR_OK;
}
```

After a successful connection, the `USER()` function should indicate the connecting client user and host name, and `CURRENT_USER()` should indicate the account whose privileges apply during the session. The latter value should be the connecting user account if no proxying occurs or the proxied account if proxying does occur.

Compile and install the plugin, then test it. First, connect as `plugin_user1`:

```
shell> mysql --user=plugin_user1 --password
Enter password: x
```

In this case, there should be no proxying:

```
mysql> SELECT USER(), CURRENT_USER(), @@proxy_user, @@external_user\G
*************************** 1. row ***************************
        USER(): plugin_user1@localhost
 CURRENT_USER(): plugin_user1@localhost
   @@proxy_user: NULL
@@external_user: NULL
```

Then connect as `plugin_user2`:

```
shell> mysql --user=plugin_user2 --password
Enter password: x
```

In this case, `plugin_user2` should be proxied to `proxied_user`:

```
mysql> SELECT USER(), CURRENT_USER(), @@proxy_user, @@external_user\G
*************************** 1. row ***************************
        USER(): plugin_user2@localhost
 CURRENT_USER(): proxied_user@localhost
   @@proxy_user: 'plugin_user2'@'localhost'
@@external_user: 'plugin_user2'@'localhost'
```

4.4.10 Writing Password-Validation Plugins

This section describes how to write a server-side password-validation plugin. The instructions are based on the source code in the `plugin/password_validation` directory of MySQL source

distributions. The `validate_password.cc` source file in that directory implements the plugin named `validate_password`.

To write a password-validation plugin, include the following header file in the plugin source file. Other MySQL or general header files might also be needed, depending on the plugin capabilities and requirements.

```
#include <mysql/plugin_validate_password.h>
```

`plugin_validate_password.h` includes `plugin.h`, so you need not include the latter file explicitly. `plugin.h` defines the `MYSQL_VALIDATE_PASSWORD_PLUGIN` server plugin type and the data structures needed to declare the plugin. `plugin_validate_password.h` defines data structures specific to password-validation plugins.

A password-validation plugin, like any MySQL server plugin, has a general plugin descriptor (see Section 4.4.2.1, “Server Plugin Library and Plugin Descriptors”). In `validate_password.cc`, the general descriptor for `validate_password` looks like this:

```
mysql_declare_plugin(validate_password)
{
  MYSQL_VALIDATE_PASSWORD_PLUGIN,     /*   type                            */
  &validate_password_descriptor,      /*   descriptor                      */
  "validate_password",                /*   name                            */
  "Oracle Corporation",               /*   author                          */
  "check password strength",          /*   description                     */
  PLUGIN_LICENSE_GPL,
  validate_password_init,             /*   init function (when loaded)     */
  validate_password_deinit,           /*   deinit function (when unloaded) */
  0x0100,                             /*   version                         */
  NULL,
  validate_password_system_variables, /*   system variables                */
  NULL,
  0,
}
mysql_declare_plugin_end;
```

The `name` member (`validate_password`) indicates the name to use for references to the plugin in statements such as `INSTALL PLUGIN` or `UNINSTALL PLUGIN`. This is also the name displayed by `INFORMATION_SCHEMA.PLUGINS` or `SHOW PLUGINS`.

The general descriptor also refers to `validate_password_system_variables`, a structure that exposes several system variables to the `SHOW VARIABLES` statement:

```
static struct st_mysql_sys_var* validate_password_system_variables[]= {
  MYSQL_SYSVAR(length),
  MYSQL_SYSVAR(number_count),
  MYSQL_SYSVAR(mixed_case_count),
  MYSQL_SYSVAR(special_char_count),
  MYSQL_SYSVAR(policy),
  MYSQL_SYSVAR(dictionary_file),
  NULL
};
```

The `validate_password_init` initialization function reads the dictionary file if one was specified, and the `validate_password_deinit` function frees data structures associated with the file.

The `validate_password_descriptor` value in the general descriptor points to the type-specific descriptor. For password-validation plugins, this descriptor has the following structure:

```
struct st_mysql_validate_password
{
  int interface_version;
```

```
  /*
    This function returns TRUE for passwords which satisfy the password
    policy (as chosen by plugin variable) and FALSE for all other
    password
  */
  int (*validate_password)(mysql_string_handle password);
  /*
    This function returns the password strength (0-100) depending
    upon the policies
  */
  int (*get_password_strength)(mysql_string_handle password);
};
```

The type-specific descriptor has these members:

- `interface_version`: By convention, type-specific plugin descriptors begin with the interface version for the given plugin type. The server checks `interface_version` when it loads the plugin to see whether the plugin is compatible with it. For password-validation plugins, the value of the `interface_version` member is `MYSQL_VALIDATE_PASSWORD_INTERFACE_VERSION` (defined in `plugin_validate_password.h`).
- `validate_password`: A function that the server calls to test whether a password satisfies the current password policy. It returns 1 if the password is okay and 0 otherwise. The argument is the password, passed as a `mysql_string_handle` value. This data type is implemented by the `mysql_string` server service. For details, see the `string_service.h` and `string_service.cc` source files in the `sql` directory.
- `get_password_strength`: A function that the server calls to assess the strength of a password. It returns a value from 0 (weak) to 100 (strong). The argument is the password, passed as a `mysql_string_handle` value.

For the `validate_password` plugin, the type-specific descriptor looks like this:

```
static struct st_mysql_validate_password validate_password_descriptor=
{
  MYSQL_VALIDATE_PASSWORD_INTERFACE_VERSION,
  validate_password,                        /* validate function          */
  get_password_strength                     /* validate strength function */
};
```

To compile and install a plugin library file, use the instructions in Section 4.4.3, “Compiling and Installing Plugin Libraries”. To make the library file available for use, install it in the plugin directory (the directory named by the `plugin_dir` system variable). For the `validate_password` plugin, it is compiled and installed when you build MySQL from source. It is also included in binary distributions. The build process produces a shared object library with a name of `validate_password.so` (the `.so` suffix might differ depending on your platform).

To register the plugin at runtime, use this statement, adjusting the `.so` suffix for your platform as necessary:

```
INSTALL PLUGIN validate_password SONAME 'validate_password.so';
```

For additional information about plugin loading, see Installing and Uninstalling Plugins.

To verify plugin installation, examine the `INFORMATION_SCHEMA.PLUGINS` table or use the `SHOW PLUGINS` statement. See Obtaining Server Plugin Information.

While the `validate_password` plugin is installed, it exposes system variables that indicate the password-checking parameters:

```
mysql> SHOW VARIABLES LIKE 'validate_password%';
```

```
+-------------------------------------+--------+
| Variable_name                       | Value  |
+-------------------------------------+--------+
| validate_password_dictionary_file   |        |
| validate_password_length            | 8      |
| validate_password_mixed_case_count  | 1      |
| validate_password_number_count      | 1      |
| validate_password_policy            | MEDIUM |
| validate_password_special_char_count | 1      |
+-------------------------------------+--------+
```

For descriptions of these variables, see Password Validation Plugin Options and Variables.

To disable the plugin after testing it, use this statement to unload it:

```
UNINSTALL PLUGIN validate_password;
```

4.4.11 Writing Protocol Trace Plugins

MySQL supports the use of protocol trace plugins: client-side plugins that implement tracing of communication between a client and the server that takes place using the client/server protocol.

4.4.11.1 Using the Test Protocol Trace Plugin

MySQL includes a test protocol trace plugin that serves to illustrate the information available from such plugins, and as a guide to writing other protocol trace plugins. To see how the test plugin works, use a MySQL source distribution; binary distributions are built with the test plugin disabled.

Enable the test protocol trace plugin by configuring MySQL with the `WITH_TEST_TRACE_PLUGIN` `CMake` option enabled. This causes the test trace plugin to be built and MySQL client programs to load it, but the plugin has no effect by default. Control the plugin using these environment variables:

- `MYSQL_TEST_TRACE_DEBUG`: Set this variable to a value other than 0 to cause the test plugin to produce diagnostic output on `stderr`.
- `MYSQL_TEST_TRACE_CRASH`: Set this variable to a value other than 0 to cause the test plugin to abort the client program if it detects an invalid trace event.

> **Caution**
>
> Diagnostic output from the test protocol trace plugin can disclose passwords and other sensitive information.

Given a MySQL installation built from source with the test plugin enabled, you can see a trace of the communication between the `mysql` client and the MySQL server as follows:

```
shell> export MYSQL_TEST_TRACE_DEBUG=1
shqll> mysql
test_trace: Test trace plugin initialized
test_trace: Starting tracing in stage CONNECTING
test_trace: stage: CONNECTING, event: CONNECTING
test_trace: stage: CONNECTING, event: CONNECTED
test_trace: stage: WAIT_FOR_INIT_PACKET, event: READ_PACKET
test_trace: stage: WAIT_FOR_INIT_PACKET, event: PACKET_RECEIVED
test_trace: packet received: 87 bytes
  0A 35 2E 37 2E 33 2D 6D  31 33 2D 64 65 62 75 67   .5.7.3-m13-debug
  2D 6C 6F 67 00 04 00 00  00 2B 7C 4F 55 3F 79 67   -log.....+|OU?yg
test_trace: 004: stage: WAIT_FOR_INIT_PACKET, event: INIT_PACKET_RECEIVED
test_trace: 004: stage: AUTHENTICATE, event: AUTH_PLUGIN
test_trace: 004: Using authentication plugin: mysql_native_password
test_trace: 004: stage: AUTHENTICATE, event: SEND_AUTH_RESPONSE
```

```
test_trace: 004: sending packet: 188 bytes
  85 A6 7F 00 00 00 00 01  21 00 00 00 00 00 00 00   .?......!.......
  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
...
mysql> quit
test_trace: 008: stage: READY_FOR_COMMAND, event: SEND_COMMAND
test_trace: 008: QUIT
test_trace: 008: stage: READY_FOR_COMMAND, event: PACKET_SENT
test_trace: 008: packet sent: 0 bytes
test_trace: 008: stage: READY_FOR_COMMAND, event: DISCONNECTED
test_trace: 008: Connection  closed
test_trace: 008: Tracing connection has ended
Bye
test_trace: Test trace plugin de-initialized
```

To disable trace output, do this:

```
shell> MYSQL_TEST_TRACE_DEBUG=
```

4.4.11.2 Using Your Own Protocol Trace Plugins

> **Note**
>
> To use your own protocol trace plugins, you must configure MySQL with the `WITH_TEST_TRACE_PLUGIN` `CMake` option *disabled* because only one protocol trace plugin can be loaded at a time and an error occurs for attempts to load a second one. If you have already built MySQL with the test protocol trace plugin enabled to see how it works, you must rebuild MySQL without it before you can use your own plugins.

This section discusses how to write a basic protocol trace plugin named `simple_trace`. This plugin provides a framework showing how to set up the client plugin descriptor and create the trace-related callback functions. In `simple_trace`, these functions are rudimentary and do little other than illustrate the arguments required. To see in detail how a trace plugin can make use of trace event information, check the source file for the test protocol trace plugin (`test_trace_plugin.cc` in the `libmysql` directory of a MySQL source distribution). However, note that the `st_mysql_client_plugin_TRACE` structure used there differs from the structures used with the usual client plugin declaration macros. In particular, the first two members are defined explicitly, not implicitly by declaration macros.

Several header files contain information relevant to protocol trace plugins:

- `client_plugin.h`: Defines the API for client plugins. This includes the client plugin descriptor and function prototypes for client plugin C API calls (see C API Client Plugin Interface).
- `plugin_trace.h`: Contains declarations for client-side plugins of type `MYSQL_CLIENT_TRACE_PLUGIN`. It also contains descriptions of the permitted protocol stages, transitions between stages, and the types of events permitted at each stage.

To write a protocol trace plugin, include the following header files in the plugin source file. Other MySQL or general header files might also be needed, depending on the plugin capabilities and requirements.

```
#include <mysql/plugin_trace.h>
#include <mysql.h>
```

`plugin_trace.h` includes `client_plugin.h`, so you need not include the latter file explicitly.

Declare the client-side plugin descriptor with the `mysql_declare_client_plugin()` and `mysql_end_client_plugin` macros (see Section 4.4.2.3, “Client Plugin Descriptors”). For the `simple_trace` plugin, the descriptor looks like this:

```
mysql_declare_client_plugin(TRACE)
  "simple_trace",                 /* plugin name */
  "Author Name",                  /* author */
  "Simple protocol trace plugin", /* description */
  {1,0,0},                        /* version = 1.0.0 */
  "GPL",                          /* license type */
  NULL,                           /* for internal use */
  plugin_init,                    /* initialization function */
  plugin_deinit,                  /* deinitialization function */
  plugin_options,                 /* option-handling function */
  trace_start,                    /* start-trace function */
  trace_stop,                     /* stop-trace function */
  trace_event                     /* event-handling function */
mysql_end_client_plugin;
```

The descriptor members from the plugin name through the option-handling function are common to all client plugin types. The members following the common members implement trace event handling.

Function members for which the plugin needs no processing can be declared as `NULL` in the descriptor, in which case you need not write any corresponding function. For illustration purposes and to show the argument syntax, the following discussion implements all functions listed in the descriptor, even though some of them do nothing,

The initialization, deinitialization, and options functions common to all client plugins are declared as follows. For a description of the arguments and return values, see Section 4.4.2.3, “Client Plugin Descriptors”.

```
static int
plugin_init(char *errbuf, size_t errbuf_len, int argc, va_list args)
{
  return 0;
}

static int
plugin_deinit()
{
  return 0;
}

static int
plugin_options(const char *option, const void *value)
{
  return 0;
}
```

The trace-specific members of the client plugin descriptor are callback functions. The following descriptions provide more detail on how they are used. Each has a first argument that is a pointer to the plugin instance in case your implementation needs to access it.

`trace_start()`: This function is called at the start of each traced connection (each connection that starts after the plugin is loaded). It is passed the connection handler and the protocol stage at which tracing starts. `trace_start()` allocates memory needed by the `trace_event()` function, if any, and returns a pointer to it. If no memory is needed, this function returns `NULL`.

```
static void*
trace_start(struct st_mysql_client_plugin_TRACE *self,
            MYSQL *conn,
            enum protocol_stage stage)
{
  struct st_trace_data *plugin_data= malloc(sizeof(struct st_trace_data));

  fprintf(stderr, "Initializing trace: stage %d\n", stage);
  if (plugin_data)
  {
```

```
    memset(plugin_data, 0, sizeof(struct st_trace_data));
    fprintf(stderr, "Trace initialized\n");
    return plugin_data;
  }
  fprintf(stderr, "Could not initialize trace\n");
  exit(1);
}
```

`trace_stop()`: This function is called when tracing of the connection ends. That usually happens when the connection is closed, but can happen earlier. For example, `trace_event()` can return a nonzero value at any time and that causes tracing of the connection to terminate. `trace_stop()` is then called even though the connection has not ended.

`trace_stop()` is passed the connection handler and a pointer to the memory allocated by `trace_start()` (`NULL` if none). If the pointer is non-`NULL`, `trace_stop()` should deallocate the memory. This function returns no value.

```
static void
trace_stop(struct st_mysql_client_plugin_TRACE *self,
           MYSQL *conn,
           void *plugin_data)
{
  fprintf(stderr, "Terminating trace\n");
  if (plugin_data)
    free(plugin_data);
}
```

`trace_event()`: This function is called for each event occurrence. It is passed a pointer to the memory allocated by `trace_start()` (`NULL` if none), the connection handler, the current protocol stage and event codes, and event data. This function returns 0 to continue tracing, nonzero if tracing should stop.

```
static int
trace_event(struct st_mysql_client_plugin_TRACE *self,
            void *plugin_data,
            MYSQL *conn,
            enum protocol_stage stage,
            enum trace_event event,
            struct st_trace_event_args args)
{
  fprintf(stderr, "Trace event received: stage %d, event %d\n", stage, event);
  if (event == TRACE_EVENT_DISCONNECTED)
    fprintf(stderr, "Connection closed\n");
  return 0;
}
```

The tracing framework shuts down tracing of the connection when the connection ends, so `trace_event()` should return nonzero only if you want to terminate tracing of the connection early. Suppose that you want to trace only connections for a certain MySQL account. After authentication, you can check the user name for the connection and stop tracing if it is not the user in whom you are interested.

For each call to `trace_event()`, the `st_trace_event_args` structure contains the event data. It has this definition:

```
struct st_trace_event_args
{
  const char           *plugin_name;
  int                   cmd;
  const unsigned char  *hdr;
  size_t                hdr_len;
  const unsigned char  *pkt;
  size_t                pkt_len;
};
```

For different event types, the `st_trace_event_args` structure contains the information described following. All lengths are in bytes. Unused members are set to `0`/`NULL`.

`AUTH_PLUGIN` event:

```
plugin_name  The name of the plugin
```

`SEND_COMMAND` event:

```
cmd          The command code
hdr          Pointer to the command packet header
hdr_len      Length of the header
pkt          Pointer to the command arguments
pkt_len      Length of the arguments
```

Other `SEND_`*`xxx`* and *`xxx`*`_RECEIVED` events:

```
pkt          Pointer to the data sent or received
pkt_len      Length of the data
```

`PACKET_SENT` event:

```
pkt_len      Number of bytes sent
```

To compile and install a plugin library file, use the instructions in Section 4.4.3, “Compiling and Installing Plugin Libraries”. To make the library file available for use, install it in the plugin directory (the directory named by the `plugin_dir` system variable).

After the plugin library file is compiled and installed in the plugin directory, you can test it easily by setting the `LIBMYSQL_PLUGINS` environment variable to the plugin name, which affects any client program that uses that variable. `mysql` is one such program:

```
shell> export LIBMYSQL_PLUGINS=simple_trace
shqll> mysql
Initializing trace: stage 0
Trace initialized
Trace event received: stage 0, event 1
Trace event received: stage 0, event 2
...
Welcome to the MySQL monitor.  Commands end with ; or \g.
Trace event received
Trace event received
...
mysql> SELECT 1;
Trace event received: stage 4, event 12
Trace event received: stage 4, event 16
...
Trace event received: stage 8, event 14
Trace event received: stage 8, event 15
+---+
| 1 |
+---+
| 1 |
+---+
1 row in set (0.00 sec)

mysql> quit
Trace event received: stage 4, event 12
Trace event received: stage 4, event 16
Trace event received: stage 4, event 3
Connection closed
Terminating trace
Bye
```

To stop the trace plugin from being loaded, do this:

```
shell> LIBMYSQL_PLUGINS=
```

It is also possible to write client programs that directly load the plugin. You can tell the client where the plugin directory is located by calling `mysql_options()` to set the `MYSQL_PLUGIN_DIR` option:

```
char *plugin_dir = "path_to_plugin_dir";

/* ... process command-line options ... */

mysql_options(&mysql, MYSQL_PLUGIN_DIR, plugin_dir);
```

Typically, the program will also accept a `--plugin-dir` option that enables users to override the default value.

Should a client program require lower-level plugin management, the client library contains functions that take an `st_mysql_client_plugin` argument. See C API Client Plugin Interface.

4.4.12 Writing Keyring Plugins

MySQL Server supports a keyring service that enables internal server components and plugins to securely store sensitive information for later retrieval. This section describes how to write a server-side keyring plugin that can be used by service functions to perform key-management operations. For general keyring information, see The MySQL Keyring.

The instructions here are based on the source code in the `plugin/keyring` directory of MySQL source distributions. The source files in that directory implement a plugin named `keyring_file` that uses a file local to the server host for data storage.

To write a keyring plugin, include the following header file in the plugin source file. Other MySQL or general header files might also be needed, depending on the plugin capabilities and requirements.

```
#include <mysql/plugin_keyring.h>
```

`plugin_keyring.h` includes `plugin.h`, so you need not include the latter file explicitly. `plugin.h` defines the `MYSQL_KEYRING_PLUGIN` server plugin type and the data structures needed to declare the plugin. `plugin_keyring.h` defines data structures specific to keyring plugins.

A keyring plugin, like any MySQL server plugin, has a general plugin descriptor (see Section 4.4.2.1, “Server Plugin Library and Plugin Descriptors”). In `keyring.cc`, the general descriptor for `keyring_file` looks like this:

```
mysql_declare_plugin(keyring_file)
{
  MYSQL_KEYRING_PLUGIN,     /* type                                  */
  &keyring_descriptor,      /* descriptor                            */
  "keyring_file",           /* name                                  */
  "Oracle Corporation",     /* author                                */
  "store/fetch authentication data to/from a flat file", /* description */
  PLUGIN_LICENSE_GPL,
  keyring_init,             /* init function (when loaded)           */
  keyring_deinit,           /* deinit function (when unloaded)       */
  0x0100,                   /* version                               */
  NULL,                     /* status variables                      */
  keyring_system_variables, /* system variables                      */
  NULL,
  0,
}
mysql_declare_plugin_end;
```

The `name` member (`keyring_file`) indicates the plugin name. This is the name displayed by `INFORMATION_SCHEMA.PLUGINS` or `SHOW PLUGINS`.

The general descriptor also refers to `keyring_system_variables`, a structure that exposes a system variable to the `SHOW VARIABLES` statement:

```
static struct st_mysql_sys_var *keyring_system_variables[]= {
  MYSQL_SYSVAR(data),
  NULL
};
```

The `keyring_init` initialization function creates the data file if it does not exist, then reads it and initializes the keystore. The `keyring_deinit` function frees data structures associated with the file.

The `keyring_descriptor` value in the general descriptor points to the type-specific descriptor. For keyring plugins, this descriptor has the following structure:

```
struct st_mysql_keyring
{
  int interface_version;
  my_bool (*mysql_key_store)(const char *key_id, const char *key_type,
                             const char* user_id, const void *key, size_t key_len);
  my_bool (*mysql_key_fetch)(const char *key_id, char **key_type,
                             const char *user_id, void **key, size_t *key_len);
  my_bool (*mysql_key_remove)(const char *key_id, const char *user_id);
  my_bool (*mysql_key_generate)(const char *key_id, const char *key_type,
                                const char *user_id, size_t key_len);
};
```

The type-specific descriptor has these members:

- `interface_version`: By convention, type-specific plugin descriptors begin with the interface version for the given plugin type. The server checks `interface_version` when it loads the plugin to see whether the plugin is compatible with it. For keyring plugins, the value of the `interface_version` member is `MYSQL_KEYRING_INTERFACE_VERSION` (defined in `plugin_keyring.h`).
- `mysql_key_store`: A function that obfuscates and stores a key in the keyring.
- `mysql_key_fetch`: A function that deobfuscates and retrieves a key from the keyring.
- `mysql_key_remove`: A function that removes a key from the keyring.
- `mysql_key_generate`: A function that generates a new random key and stores it in the keyring.

For the `keyring_file` plugin, the type-specific descriptor looks like this:

```
static struct st_mysql_keyring keyring_descriptor=
{
  MYSQL_KEYRING_INTERFACE_VERSION,
  mysql_key_store,
  mysql_key_fetch,
  mysql_key_remove,
  mysql_key_generate
};
```

The `mysql_key_`*`xxx`* functions implemented by a keyring plugin are analogous to the `my_key_`*`xxx`* functions exposed by the keyring service API. For example, the `mysql_key_store` plugin function is analogous to the `my_key_store` keyring service function. For information about the arguments to keyring service functions and how they are used, see The Keyring Service.

To compile and install a plugin library file, use the instructions in Section 4.4.3, “Compiling and Installing Plugin Libraries”. To make the library file available for use, install it in the plugin directory (the directory

named by the `plugin_dir` system variable). For the `keyring_file` plugin, it is compiled and installed when you build MySQL from source. It is also included in binary distributions. The build process produces a shared object library with a name of `keyring_file.so` (the `.so` suffix might differ depending on your platform).

Keyring plugins typically are loaded early during the server startup process so that they are available to built-in plugins and storage engines that might depend on them. For `keyring_file`, use these lines in the server `my.cnf` file, adjusting the `.so` suffix for your platform as necessary:

```
[mysqld]
early-plugin-load=keyring_file.so
```

For additional information about plugin loading, see Installing and Uninstalling Plugins.

To verify plugin installation, examine the `INFORMATION_SCHEMA.PLUGINS` table or use the `SHOW PLUGINS` statement (see Obtaining Server Plugin Information). For example:

```
mysql> SELECT PLUGIN_NAME, PLUGIN_STATUS
       FROM INFORMATION_SCHEMA.PLUGINS
       WHERE PLUGIN_NAME LIKE 'keyring%';
+--------------+---------------+
| PLUGIN_NAME  | PLUGIN_STATUS |
+--------------+---------------+
| keyring_file | ACTIVE        |
+--------------+---------------+
```

While the `keyring_file` plugin is installed, it exposes a system variable that indicates the location of the data file it uses for secure information storage:

```
mysql> SHOW VARIABLES LIKE 'keyring_file%';
+-------------------+---------------------------------+
| Variable_name     | Value                           |
+-------------------+---------------------------------+
| keyring_file_data | /usr/local/mysql/keyring/keyring |
+-------------------+---------------------------------+
```

For a description of the `keyring_file_data` variable, see Server System Variables.

To disable the plugin after testing it, restart the server without an `--early-plugin-load` option that names the plugin.

Chapter 5 MySQL Services for Plugins

MySQL server plugins have access to server “plugin services.” The plugin services interface exposes server functionality that plugins can call. It complements the plugin API and has these characteristics:

- Services enable plugins to access code inside the server using ordinary function calls. Services are also available to loadable functions.
- Services are portable and work on multiple platforms.
- The interface includes a versioning mechanism so that service versions supported by the server can be checked at load time against plugin versions. Versioning protects against incompatibilities between the version of a service that the server provides and the version of the service expected or required by a plugin.
- For information about plugins for testing plugin services, see the Plugins for Testing Plugin Services section of the MySQL Server Doxygen documentation, available at https://dev.mysql.com/doc/index-other.html.

The plugin services interface differs from the plugin API as follows:

- The plugin API enables plugins to be used by the server. The calling initiative lies with the server to invoke plugins. This enables plugins to extend server functionality or register to receive notifications about server processing.
- The plugin services interface enables plugins to call code inside the server. The calling initiative lies with plugins to invoke service functions. This enables functionality already implemented in the server to be used by many plugins; they need not individually implement it themselves.

To determine what services exist and what functions they provide, look in the `include/mysql` directory of a MySQL source distribution. The relevant files are:

- `plugin.h` includes `services.h`, which is the “umbrella” header that includes all available service-specific header files.
- Service-specific headers have names of the form `service_`*`xxx`*`.h`.

Each service-specific header should contain comments that provide full usage documentation for a given service, including what service functions are available, their calling sequences, and return values.

For developers who wish to modify the server to add a new service, see MySQL Internals: MySQL Services for Plugins.

Available services include the following:

- `locking_service`: A service that implements locks with three attributes: Lock namespace, lock name, and lock mode. This locking interface is accessible at two levels: 1) At the SQL level, as a set of loadable functions that each map onto calls to the service routines; 2) As a C language interface, callable as a plugin service from server plugins or loadable functions. For more information, see The Locking Service.
- `my_plugin_log_service`: A service that enables plugins to report errors and specify error messages. The server writes the messages to its error log.
- `my_snprintf`: A string-formatting service that produces consistent results across platforms.
- `my_thd_scheduler`: A service for plugins to select a thread scheduler.

- `mysql_keyring`: A service for keyring storage, accessible at two levels: 1) At the SQL level, as a set of loadable functions that each map onto calls to the service routines; 2) As a C language interface, callable as a plugin service from server plugins or loadable functions. For more information, see The Keyring Service.
- `mysql_password_policy`: A service for password validation and strength checking.
- `mysql_string`: A service for string manipulation.
- `security_context`: A service that enables plugins to examine or manipulate thread security contexts. This service provides setter and getter routines to access attributes of the server `Security_context` class, which includes attributes such as operating system user and host, authenticated user and host, and client IP address.
- `thd_alloc`: A memory-allocation service.
- `thd_wait`: A service for plugins to report when they are going to sleep or stall.

The remainder of this section describes how a plugin uses server functionality that is available as a service. See also the source for the “daemon” example plugin, which uses the `my_snprintf` service. Within a MySQL source distribution, that plugin is located in the `plugin/daemon_example` directory.

To use a service or services from within a plugin, the plugin source file must include the `plugin.h` header file to access service-related information:

```
#include <mysql/plugin.h>
```

This does not represent any additional setup cost. A plugin must include that file anyway because it contains definitions and structures that every plugin needs.

To access a service, a plugin calls service functions like any other function. For example, to format a string into a buffer for printing, call the `my_snprintf()` function provided by the service of the same name:

```
char buffer[BUFFER_SIZE];

my_snprintf(buffer, sizeof(buffer), format_string, argument_to_format, ...);
```

To report an error that the server will write to it error log, first choose an error level. `mysql/service_my_plugin_log.h` defines these levels:

```
enum plugin_log_level
{
  MY_ERROR_LEVEL,
  MY_WARNING_LEVEL,
  MY_INFORMATION_LEVEL
};
```

Then invoke `my_plugin_log_message()`:

```
int my_plugin_log_message(MYSQL_PLUGIN *plugin, enum plugin_log_level level,
                          const char *format, ...);
```

For example:

```
my_plugin_log_message(plugin_ptr, MY_ERROR_LEVEL, "Cannot initialize plugin");
```

Some services *for* plugins may be provided *by* plugins and thus are available only if the service-providing plugin is loaded. Any MySQL component that uses such a service should check whether the service is available.

When you build your plugin, use the `-lmysqlservices` flag at link time to link in the `libmysqlservices` library. For example, for `CMake`, put this in the top-level `CMakeLists.txt` file:

```
FIND_LIBRARY(MYSQLSERVICES_LIB mysqlservices
 PATHS "${MYSQL_SRCDIR}/libservices" NO_DEFAULT_PATH)
```

Put this in the `CMakeLists.txt` file in the directory containing the plugin source:

```
# the plugin needs the mysql services library for error logging
TARGET_LINK_LIBRARIES (your_plugin_library_name ${MYSQLSERVICES_LIB})
```

Chapter 6 Adding Functions to MySQL

Table of Contents

There are three ways to add a new function to MySQL:

- Create a stored function (a type of stored object). A stored function is written using SQL statements rather than by compiling object code. The syntax for writing stored functions is not covered here. See Using Stored Routines.
- Create a native (built-in) MySQL function. A native function is added by modifying the MySQL source code to be compiled into the `mysqld` server and become available on a permanent basis. See Section 6.1, “Adding a Native Function”.
- Use the loadable function interface. A loadable function is compiled as a library file and then loaded and unloaded from the server dynamically using the `CREATE FUNCTION` and `DROP FUNCTION` statements. See Section 6.2, “Adding a Loadable Function”.

> **Note**
>
> Loadable functions previously were known as user-defined functions (UDFs). That terminology was something of a misnomer because “user-defined” also can apply to stored functions written using SQL and native functions added by modifying the server source code.

Each method of creating compiled functions has advantages and disadvantages:

- Adding a native function requires modifying a source distribution. Adding a loadable function does not; it can be added to a binary MySQL distribution with no access to MySQL source necessary.
- A loadable function is contained in an object file that you must install in addition to the server itself. For a function compiled into the server, that is unnecessary.
- If you upgrade your MySQL distribution, you can continue to use previously installed loadable functions, unless you upgrade to a newer MySQL version for which the loadable function interface changes. For native functions, you must repeat your source code modifications each time you upgrade.

Regardless of the method used to add a function, it can be invoked in SQL statements just like native functions such as `ABS()` or `SOUNDEX()`.

For the rules describing how the server interprets references to different kinds of functions, see Function Name Parsing and Resolution.

The following sections describe features of the loadable function interface, provide instructions for writing loadable functions, discuss security precautions that MySQL takes to prevent loadable function misuse, and describe how to add native MySQL functions.

For example source code that illustrates how to write loadable functions, take a look at the `sql/udf_example.cc` file that is provided in MySQL source distributions.

6.1 Adding a Native Function

To add a native MySQL function, use the procedure described here, which requires that you use a source distribution. You cannot add native functions to a binary distribution because it is necessary to modify MySQL source code and compile MySQL from the modified source. If you migrate to another version of MySQL (for example, when a new version is released), you must repeat the procedure with the new version.

If the native function will be referred to in statements that will be replicated to replicas, you must ensure that every replica also has the function available. Otherwise, replication will fail on the replicas when they attempt to invoke the function.

To add a native function, follow these steps to modify source files in the `sql` directory:

1. Create a subclass for the function in `item_create.cc`:

   - If the function takes a fixed number of arguments, create a subclass of `Create_func_arg0`, `Create_func_arg1`, `Create_func_arg2`, or `Create_func_arg3`, respectively, depending on whether the function takes zero, one, two, or three arguments. For examples, see the `Create_func_uuid`, `Create_func_abs`, `Create_func_pow`, and `Create_func_lpad` classes.
   - If the function takes a variable number of arguments, create a subclass of `Create_native_func`. For an example, see `Create_func_concat`.

2. To provide a name by which the function can be referred to in SQL statements, register the name in `item_create.cc` by adding a line to this array:

   ```
   static Native_func_registry func_array[]
   ```

   You can register several names for the same function. For example, see the lines for `"LCASE"` and `"LOWER"`, which are aliases for `Create_func_lcase`.

3. In `item_func.h`, declare a class inheriting from `Item_num_func` or `Item_str_func`, depending on whether your function returns a number or a string.

4. In `item_func.cc`, add one of the following declarations, depending on whether you are defining a numeric or string function:

   ```
   double   Item_func_newname::val()
   longlong Item_func_newname::val_int()
   String  *Item_func_newname::Str(String *str)
   ```

   If you inherit your object from any of the standard items (like `Item_num_func`), you probably only have to define one of these functions and let the parent object take care of the other functions. For example, the `Item_str_func` class defines a `val()` function that executes `atof()` on the value returned by `::str()`.

5. If the function is nondeterministic, include the following statement in the item constructor to indicate that function results should not be cached:

   ```
   current_thd->lex->safe_to_cache_query=0;
   ```

   A function is nondeterministic if, given fixed values for its arguments, it can return different results for different invocations.

6. You should probably also define the following object function:

   ```
   void Item_func_newname::fix_length_and_dec()
   ```

   This function should at least calculate `max_length` based on the given arguments. `max_length` is the maximum number of characters the function may return. This function should also set `maybe_null`

`= 0` if the main function cannot return a `NULL` value. The function can check whether any of the function arguments can return `NULL` by checking the arguments' `maybe_null` variable. Look at `Item_func_mod::fix_length_and_dec` for a typical example of how to do this.

All functions must be thread-safe. In other words, do not use any global or static variables in the functions without protecting them with mutexes.

If you want to return `NULL` from `::val()`, `::val_int()`, or `::str()`, you should set `null_value` to 1 and return 0.

For `::str()` object functions, these additional considerations apply:

- The `String *str` argument provides a string buffer that may be used to hold the result. (For more information about the `String` type, take a look at the `sql_string.h` file.)
- The `::str()` function should return the string that holds the result, or `(char*) 0` if the result is `NULL`.
- All current string functions try to avoid allocating any memory unless absolutely necessary!

6.2 Adding a Loadable Function

The MySQL interface for loadable functions provides the following features and capabilities:

- Functions can return string, integer, or real values and can accept arguments of those same types.
- You can define simple functions that operate on a single row at a time, or aggregate functions that operate on groups of rows.
- Information is provided to functions that enables them to check the number, types, and names of the arguments passed to them.
- You can tell MySQL to coerce arguments to a given type before passing them to a function.
- You can indicate that a function returns `NULL` or that an error occurred.

For the loadable function mechanism to work, functions must be written in C or C++ and your operating system must support dynamic loading. MySQL source distributions include a file `sql/udf_example.cc` that defines five loadable function interface functions. Consult this file to see how loadable function calling conventions work. The `include/mysql_com.h` header file defines loadable function-related symbols and data structures, although you need not include this header file directly; it is included by `mysql.h`.

A loadable function contains code that becomes part of the running server, so when you write a loadable function, you are bound by any and all constraints that apply to writing server code. For example, you may have problems if you attempt to use functions from the `libstdc++` library. These constraints may change in future versions of the server, so it is possible that server upgrades will require revisions to loadable functions that were originally written for older servers. For information about these constraints, see MySQL Source-Configuration Options, and Dealing with Problems Compiling MySQL.

To be able to use loadable functions, you must link `mysqld` dynamically. If you want to use a loadable function that needs to access symbols from `mysqld` (for example, the `metaphone` function in `sql/udf_example.cc` uses `default_charset_info`), you must link the program with `-rdynamic` (see `man dlopen`).

For each function that you want to use in SQL statements, you should define corresponding C (or C++) functions. In the following discussion, the name “xxx” is used for an example function name. To distinguish between SQL and C/C++ usage, `XXX()` (uppercase) indicates an SQL function call, and `xxx()` (lowercase) indicates a C/C++ function call.

> **Note**
>
> When using C++, encapsulate your C functions within this construct:
>
> ```
> extern "C" { ... }
> ```
>
> This ensures that your C++ function names remain readable in the completed function.

Loadable Function Interface Functions

The following list describes the C/C++ functions that you write to implement the interface for a function named `XXX()`. The main function, `xxx()`, is required. In addition, a loadable function requires at least one of the other functions described here, for reasons discussed in Loadable Function Security Precautions.

- `xxx()`

  The main function. This is where the function result is computed. The correspondence between the SQL function data type and the return type of your C/C++ function is shown here.

SQL Type	C/C++ Type
`STRING`	`char *`
`INTEGER`	`long long`
`REAL`	`double`

  It is also possible to declare a `DECIMAL` function, but the value is returned as a string, so you should write the function as though it were a `STRING` function. `ROW` functions are not implemented.

- `xxx_init()`

  The initialization function for `xxx()`. If present, it can be used for the following purposes:

  - To check the number of arguments to `XXX()`.
  - To verify that the arguments are of a required type or, alternatively, to tell MySQL to coerce arguments to the required types when the main function is called.
  - To allocate any memory required by the main function.
  - To specify the maximum length of the result.
  - To specify (for `REAL` functions) the maximum number of decimal places in the result.
  - To specify whether the result can be `NULL`.

- `xxx_deinit()`

  The deinitialization function for `xxx()`. If present, it should deallocate any memory allocated by the initialization function.

When an SQL statement invokes `XXX()`, MySQL calls the initialization function `xxx_init()` to let it perform any required setup, such as argument checking or memory allocation. If `xxx_init()` returns an error, MySQL aborts the SQL statement with an error message and does not call the main or deinitialization functions. Otherwise, MySQL calls the main function `xxx()` once for each row. After all rows have been processed, MySQL calls the deinitialization function `xxx_deinit()` so that it can perform any required cleanup.

For aggregate functions that work like `SUM()`, you must also provide the following functions:

- `xxx_clear()`

  Reset the current aggregate value but do not insert the argument as the initial aggregate value for a new group.

- `xxx_add()`

  Add the argument to the current aggregate value.

MySQL handles aggregate loadable functions as follows:

1. Call `xxx_init()` to let the aggregate function allocate any memory it needs for storing results.
2. Sort the table according to the `GROUP BY` expression.
3. Call `xxx_clear()` for the first row in each new group.
4. Call `xxx_add()` for each row that belongs in the same group.
5. Call `xxx()` to get the result for the aggregate when the group changes or after the last row has been processed.
6. Repeat steps 3 to 5 until all rows has been processed
7. Call `xxx_deinit()` to let the function free any memory it has allocated.

All functions must be thread-safe. This includes not just the main function, but the initialization and deinitialization functions as well, and also the additional functions required by aggregate functions. A consequence of this requirement is that you are not permitted to allocate any global or static variables that change! If you need memory, you must it in `xxx_init()` and free it in `xxx_deinit()`.

Loadable Function Calling Sequences for Simple Functions

This section describes the different interface functions that you must define to create a simple loadable function. For information about the order in which MySQL calls these functions, see Loadable Function Interface Functions.

The main `xxx()` function should be declared as shown in this section. Note that the return type and parameters differ, depending on whether you declare the SQL function `XXX()` to return `STRING`, `INTEGER`, or `REAL` in the `CREATE FUNCTION` statement:

For `STRING` functions:

```
char *xxx(UDF_INIT *initid, UDF_ARGS *args,
          char *result, unsigned long *length,
          char *is_null, char *error);
```

For `INTEGER` functions:

```
long long xxx(UDF_INIT *initid, UDF_ARGS *args,
              char *is_null, char *error);
```

For `REAL` functions:

```
double xxx(UDF_INIT *initid, UDF_ARGS *args,
              char *is_null, char *error);
```

`DECIMAL` functions return string values and are declared the same way as `STRING` functions. `ROW` functions are not implemented.

Declare the initialization and deinitialization functions like this:

```
my_bool xxx_init(UDF_INIT *initid, UDF_ARGS *args, char *message);

void xxx_deinit(UDF_INIT *initid);
```

The `initid` parameter is passed to all three functions. It points to a `UDF_INIT` structure that is used to communicate information between functions. The `UDF_INIT` structure members follow. The initialization function should fill in any members that it wishes to change. (To use the default for a member, leave it unchanged.)

- `my_bool maybe_null`

  `xxx_init()` should set `maybe_null` to `1` if `xxx()` can return `NULL`. The default value is `1` if any of the arguments are declared `maybe_null`.

- `unsigned int decimals`

  The number of decimal digits to the right of the decimal point. The default value is the maximum number of decimal digits in the arguments passed to the main function. For example, if the function is passed `1.34`, `1.345`, and `1.3`, the default would be 3, because `1.345` has 3 decimal digits.

  For arguments that have no fixed number of decimals, the `decimals` value is set to 31, which is 1 more than the maximum number of decimals permitted for the `DECIMAL`, `FLOAT`, and `DOUBLE` data types. This value is available as the constant `NOT_FIXED_DEC` in the `mysql_com.h` header file.

  A `decimals` value of 31 is used for arguments in cases such as a `FLOAT` or `DOUBLE` column declared without an explicit number of decimals (for example, `FLOAT` rather than `FLOAT(10,3)`) and for floating-point constants such as `1345E-3`. It is also used for string and other nonnumber arguments that might be converted within the function to numeric form.

  The value to which the `decimals` member is initialized is only a default. It can be changed within the function to reflect the actual calculation performed. The default is determined such that the largest number of decimals of the arguments is used. If the number of decimals is `NOT_FIXED_DEC` for even one of the arguments, that is the value used for `decimals`.

- `unsigned int max_length`

  The maximum length of the result. The default `max_length` value differs depending on the result type of the function. For string functions, the default is the length of the longest argument. For integer functions, the default is 21 digits. For real functions, the default is 13 plus the number of decimal digits indicated by `initid->decimals`. (For numeric functions, the length includes any sign or decimal point characters.)

  If you want to return a blob value, you can set `max_length` to 65KB or 16MB. This memory is not allocated, but the value is used to decide which data type to use if there is a need to temporarily store the data.

- `char *ptr`

  A pointer that the function can use for its own purposes. For example, functions can use `initid->ptr` to communicate allocated memory among themselves. `xxx_init()` should allocate the memory and assign it to this pointer:

  ```
  initid->ptr = allocated_memory;
  ```

  In `xxx()` and `xxx_deinit()`, refer to `initid->ptr` to use or deallocate the memory.

- `my_bool const_item`

  `xxx_init()` should set `const_item` to `1` if `xxx()` always returns the same value and to `0` otherwise.

Loadable Function Calling Sequences for Aggregate Functions

This section describes the different interface functions that you need to define when you create an aggregate loadable function. For information about the order in which MySQL calls these functions, see Loadable Function Interface Functions.

- `xxx_reset()`

  This function is called when MySQL finds the first row in a new group. It should reset any internal summary variables and then use the given `UDF_ARGS` argument as the first value in your internal summary value for the group. Declare `xxx_reset()` as follows:

  ```
  void xxx_reset(UDF_INIT *initid, UDF_ARGS *args,
                 char *is_null, char *error);
  ```

  `xxx_reset()` is not needed or used in MySQL 5.7, in which the loadable function interface uses `xxx_clear()` instead. However, you can define both `xxx_reset()` and `xxx_clear()` if you want to have your function work with older versions of the server. (If you do include both functions, the `xxx_reset()` function in many cases can be implemented internally by calling `xxx_clear()` to reset all variables, and then calling `xxx_add()` to add the `UDF_ARGS` argument as the first value in the group.)

- `xxx_clear()`

  This function is called when MySQL needs to reset the summary results. It is called at the beginning for each new group but can also be called to reset the values for a query where there were no matching rows. Declare `xxx_clear()` as follows:

  ```
  void xxx_clear(UDF_INIT *initid, char *is_null, char *error);
  ```

  `is_null` is set to point to `CHAR(0)` before calling `xxx_clear()`.

  If something went wrong, you can store a value in the variable to which the `error` argument points. `error` points to a single-byte variable, not to a string buffer.

  `xxx_clear()` is required by MySQL 5.7.

- `xxx_add()`

  This function is called for all rows that belong to the same group. You should use it to add the value in the `UDF_ARGS` argument to your internal summary variable.

  ```
  void xxx_add(UDF_INIT *initid, UDF_ARGS *args,
               char *is_null, char *error);
  ```

The `xxx()` function for an aggregate loadable function should be declared the same way as for a nonaggregate loadable function. See Loadable Function Calling Sequences for Simple Functions.

For an aggregate loadable function, MySQL calls the `xxx()` function after all rows in the group have been processed. You should normally never access its `UDF_ARGS` argument here but instead return a value based on your internal summary variables.

Return value handling in `xxx()` should be done the same way as for a nonaggregate loadable function. See Loadable Function Return Values and Error Handling.

The `xxx_reset()` and `xxx_add()` functions handle their `UDF_ARGS` argument the same way as functions for nonaggregate loadable functions. See Loadable Function Argument Processing.

The pointer arguments to `is_null` and `error` are the same for all calls to `xxx_reset()`, `xxx_clear()`, `xxx_add()` and `xxx()`. You can use this to remember that you got an error or whether the `xxx()` function should return `NULL`. You should not store a string into `*error`! `error` points to a single-byte variable, not to a string buffer.

`*is_null` is reset for each group (before calling `xxx_clear()`). `*error` is never reset.

If `*is_null` or `*error` are set when `xxx()` returns, MySQL returns `NULL` as the result for the group function.

Loadable Function Argument Processing

The `args` parameter points to a `UDF_ARGS` structure that has the members listed here:

- `unsigned int arg_count`

  The number of arguments. Check this value in the initialization function if you require your function to be called with a particular number of arguments. For example:

  ```
  if (args->arg_count != 2)
  {
      strcpy(message,"XXX() requires two arguments");
      return 1;
  }
  ```

  For other `UDF_ARGS` member values that are arrays, array references are zero-based. That is, refer to array members using index values from 0 to `args->arg_count` − 1.

- `enum Item_result *arg_type`

  A pointer to an array containing the types for each argument. The possible type values are `STRING_RESULT`, `INT_RESULT`, `REAL_RESULT`, and `DECIMAL_RESULT`.

  To make sure that arguments are of a given type and return an error if they are not, check the `arg_type` array in the initialization function. For example:

  ```
  if (args->arg_type[0] != STRING_RESULT ||
      args->arg_type[1] != INT_RESULT)
  {
      strcpy(message,"XXX() requires a string and an integer");
      return 1;
  }
  ```

  Arguments of type `DECIMAL_RESULT` are passed as strings, so you handle them the same way as `STRING_RESULT` values.

As an alternative to requiring your function's arguments to be of particular types, you can use the initialization function to set the `arg_type` elements to the types you want. This causes MySQL to coerce arguments to those types for each call to `xxx()`. For example, to specify that the first two arguments should be coerced to string and integer, respectively, do this in `xxx_init()`:

```
args->arg_type[0] = STRING_RESULT;
args->arg_type[1] = INT_RESULT;
```

Exact-value decimal arguments such as `1.3` or `DECIMAL` column values are passed with a type of `DECIMAL_RESULT`. However, the values are passed as strings. To receive a number, use the initialization function to specify that the argument should be coerced to a `REAL_RESULT` value:

```
args->arg_type[2] = REAL_RESULT;
```

- `char **args`

  `args->args` communicates information to the initialization function about the general nature of the arguments passed to your function. For a constant argument `i`, `args->args[i]` points to the argument value. (See later for instructions on how to access the value properly.) For a nonconstant argument, `args->args[i]` is `0`. A constant argument is an expression that uses only constants, such as `3` or `4*7-2` or `SIN(3.14)`. A nonconstant argument is an expression that refers to values that may change from row to row, such as column names or functions that are called with nonconstant arguments.

  For each invocation of the main function, `args->args` contains the actual arguments that are passed for the row currently being processed.

  If argument `i` represents `NULL`, `args->args[i]` is a null pointer (0). If the argument is not `NULL`, functions can refer to it as follows:

  - An argument of type `STRING_RESULT` is given as a string pointer plus a length, to enable handling of binary data or data of arbitrary length. The string contents are available as `args->args[i]` and the string length is `args->lengths[i]`. Do not assume that the string is null-terminated.
  - For an argument of type `INT_RESULT`, you must cast `args->args[i]` to a `long long` value:

    ```
    long long int_val;
    int_val = *((long long*) args->args[i]);
    ```

  - For an argument of type `REAL_RESULT`, you must cast `args->args[i]` to a `double` value:

    ```
    double    real_val;
    real_val = *((double*) args->args[i]);
    ```

  - For an argument of type `DECIMAL_RESULT`, the value is passed as a string and should be handled like a `STRING_RESULT` value.
  - `ROW_RESULT` arguments are not implemented.

- `unsigned long *lengths`

  For the initialization function, the `lengths` array indicates the maximum string length for each argument. You should not change these. For each invocation of the main function, `lengths` contains the actual lengths of any string arguments that are passed for the row currently being processed. For arguments of types `INT_RESULT` or `REAL_RESULT`, `lengths` still contains the maximum length of the argument (as for the initialization function).

- `char *maybe_null`

  For the initialization function, the `maybe_null` array indicates for each argument whether the argument value might be null (0 if no, 1 if yes).

- `char **attributes`

  `args->attributes` communicates information about the names of the function arguments. For argument `i`, the attribute name is available as a string in `args->attributes[i]` and the attribute length is `args->attribute_lengths[i]`. Do not assume that the string is null-terminated.

  By default, the name of a function argument is the text of the expression used to specify the argument. For loadable functions, an argument may also have an optional `[AS] alias_name` clause, in which case the argument name is *`alias_name`*. The `attributes` value for each argument thus depends on whether an alias was given.

  Suppose that a loadable function `my_udf()` is invoked as follows:

```
SELECT my_udf(expr1, expr2 AS alias1, expr3 alias2);
```

  In this case, the `attributes` and `attribute_lengths` arrays will have these values:

```
args->attributes[0] = "expr1"
args->attribute_lengths[0] = 5

args->attributes[1] = "alias1"
args->attribute_lengths[1] = 6

args->attributes[2] = "alias2"
args->attribute_lengths[2] = 6
```

- `unsigned long *attribute_lengths`

  The `attribute_lengths` array indicates the length of each argument name.

Loadable Function Return Values and Error Handling

The initialization function should return `0` if no error occurred and `1` otherwise. If an error occurs, `xxx_init()` should store a null-terminated error message in the `message` parameter. The message is returned to the client. The message buffer is `MYSQL_ERRMSG_SIZE` characters long. Try to keep the message to less than 80 characters so that it fits the width of a standard terminal screen.

The return value of the main function `xxx()` is the function value, for `long long` and `double` functions. A string function should return a pointer to the result and set `*length` to the length (in bytes) of the return value. For example:

```
memcpy(result, "result string", 13);
*length = 13;
```

MySQL passes a buffer to the `xxx()` function using the `result` parameter. This buffer is sufficiently long to hold 255 characters, which can be multibyte characters. The `xxx()` function can store the result in this buffer if it fits, in which case the return value should be a pointer to the buffer. If the function stores the result in a different buffer, it should return a pointer to that buffer.

If your string function does not use the supplied buffer (for example, if it needs to return a string longer than 255 characters), you must allocate the space for your own buffer with `malloc()` in the `xxx_init()` function or the `xxx()` function and free it in your `xxx_deinit()` function. You can store the allocated memory in the `ptr` slot in the `UDF_INIT` structure for reuse by future `xxx()` calls. See Loadable Function Calling Sequences for Simple Functions.

To indicate a return value of `NULL` in the main function, set `*is_null` to `1`:

```
*is_null = 1;
```

To indicate an error return in the main function, set `*error` to `1`:

```
*error = 1;
```

If `xxx()` sets `*error` to `1` for any row, the function value is `NULL` for the current row and for any subsequent rows processed by the statement in which `XXX()` was invoked. (`xxx()` is not even called for subsequent rows.)

Loadable Function Compiling and Installing

Files implementing loadable functions must be compiled and installed on the host where the server runs. The process is described here for the example loadable function file `sql/udf_example.cc` that is included in MySQL source distributions. For additional information about loadable function installation, see Installing and Uninstalling Loadable Functions.

If a loadable function will be referred to in statements that will be replicated to replicas, you must ensure that every replica also has the function available. Otherwise, replication fails on the replicas when they attempt to invoke the function.

The `udf_example.cc` file contains the following functions:

- `metaphon()` returns a metaphon string of the string argument. This is something like a soundex string, but it is more tuned for English.
- `myfunc_double()` returns the sum of the ASCII values of the characters in its arguments, divided by the sum of the length of its arguments.
- `myfunc_int()` returns the sum of the length of its arguments.
- `sequence([const int])` returns a sequence starting from the given number or 1 if no number has been given.
- `lookup()` returns the IP address for a host name.
- `reverse_lookup()` returns the host name for an IP address. The function may be called either with a single string argument of the form `'xxx.xxx.xxx.xxx'` or with four numbers.
- `avgcost()` returns an average cost. This is an aggregate function.

On Unix and Unix-like systems, compile loadable functions using the following procedure:

A dynamically loadable file should be compiled as a sharable library file, using a command something like this:

```
gcc -shared -o udf_example.so udf_example.cc
```

If you are using `gcc` with `CMake` (which is how MySQL itself is configured), you should be able to create `udf_example.so` with a simpler command:

```
make udf_example
```

After compiling a shared object containing loadable functions, you must install it and tell MySQL about it. Compiling a shared object from `udf_example.cc` using `gcc` directly produces a file named `udf_example.so`. Copy the shared object to the server's plugin directory and name it `udf_example.so`. This directory is given by the value of the `plugin_dir` system variable.

On some systems, the `ldconfig` program that configures the dynamic linker does not recognize a shared object unless its name begins with `lib`. In this case you should rename a file such as `udf_example.so` to `libudf_example.so`.

On Windows, compile loadable functions using the following procedure:

1. Obtain a MySQL source distribution. See How to Get MySQL.
2. Obtain the `CMake` build utility, if necessary, from http://www.cmake.org. (Version 2.6 or later is required).
3. In the source tree, look in the `sql` directory for files named `udf_example.def` and `udf_example.cc`. Copy both files from this directory to your working directory.
4. Create a `CMake makefile` (`CMakeLists.txt`) with these contents:

```
PROJECT(udf_example)

# Path for MySQL include directory
INCLUDE_DIRECTORIES("c:/mysql/include")

ADD_DEFINITIONS("-DHAVE_DLOPEN")
ADD_LIBRARY(udf_example MODULE udf_example.cc udf_example.def)
TARGET_LINK_LIBRARIES(udf_example wsock32)
```

5. Create the VC project and solution files, substituting an appropriate *`generator`* value:

```
cmake -G "generator"
```

   Invoking `cmake --help` shows you a list of valid generators.

6. Create `udf_example.dll`:

```
devenv udf_example.sln /build Release
```

On all platforms, after the shared library file has been copied to the `plugin_dir` directory, notify `mysqld` about the new functions with the following statements. The file name suffix differs per platform (for example, `.so` for Unix and Unix-like systems, `.dll` for Windows), so adjust the `.so` suffix for your platform as necessary.

```
CREATE FUNCTION metaphon RETURNS STRING
  SONAME 'udf_example.so';
CREATE FUNCTION myfunc_double RETURNS REAL
  SONAME 'udf_example.so';
CREATE FUNCTION myfunc_int RETURNS INTEGER
  SONAME 'udf_example.so';
CREATE FUNCTION sequence RETURNS INTEGER
  SONAME 'udf_example.so';
CREATE FUNCTION lookup RETURNS STRING
  SONAME 'udf_example.so';
CREATE FUNCTION reverse_lookup RETURNS STRING
  SONAME 'udf_example.so';
CREATE AGGREGATE FUNCTION avgcost RETURNS REAL
  SONAME 'udf_example.so';
```

Once installed, a function remains installed until it is uninstalled.

To remove functions, use `DROP FUNCTION`:

```
DROP FUNCTION metaphon;
DROP FUNCTION myfunc_double;
DROP FUNCTION myfunc_int;
```

```
DROP FUNCTION sequence;
DROP FUNCTION lookup;
DROP FUNCTION reverse_lookup;
DROP FUNCTION avgcost;
```

The `CREATE FUNCTION` and `DROP FUNCTION` statements update the `mysql.func` system table that serves as a loadable function registry. These statements require the `INSERT` and `DELETE` privilege, respectively, for the `mysql` database.

During the normal startup sequence, the server loads functions registered in the `mysql.func` table. If the server is started with the `--skip-grant-tables` option, functions registered in the table are not loaded and are unavailable.

Loadable Function Security Precautions

MySQL takes several measures to prevent misuse of loadable functions.

Loadable function library files cannot be placed in arbitrary directories. They must be located in the server's plugin directory. This directory is given by the value of the `plugin_dir` system variable.

To use `CREATE FUNCTION` or `DROP FUNCTION`, you must have the `INSERT` or `DELETE` privilege, respectively, for the `mysql` database. This is necessary because those statements add and delete rows from the `mysql.func` table.

Loadable functions should have at least one symbol defined in addition to the `xxx` symbol that corresponds to the main `xxx()` function. These auxiliary symbols correspond to the `xxx_init()`, `xxx_deinit()`, `xxx_reset()`, `xxx_clear()`, and `xxx_add()` functions. `mysqld` also supports an `--allow-suspicious-udfs` option that controls whether Loadable functions that have only an `xxx` symbol can be loaded. By default, the option is disabled, to prevent attempts at loading functions from shared library files other than those containing legitimate Loadable functions. If you have older Loadable functions that contain only the `xxx` symbol and that cannot be recompiled to include an auxiliary symbol, it may be necessary to specify the `--allow-suspicious-udfs` option. Otherwise, you should avoid enabling it.

Chapter 7 Porting MySQL

Before attempting to port MySQL to other operating systems, check the list of currently supported operating systems first. See https://www.mysql.com/support/supportedplatforms/database.html.

> **Note**
>
> If you create a new port of MySQL, you are free to copy and distribute it under the GPL license, but it does not make you a copyright holder of MySQL.

A working POSIX thread library is needed for the server.

To build MySQL from source, your system must satisfy the tool requirements listed at Installing MySQL from Source.

> **Important**
>
> If you are trying to build MySQL 5.7 with `icc` on the IA64 platform, and need support for NDB Cluster, you should first ensure that you are using `icc` version 9.1.043 or later. (For details, see Bug #21875.)

If you run into problems with a new port, you may have to do some debugging of MySQL! See Debugging a MySQL Server.

> **Note**
>
> Before you start debugging `mysqld`, first get the test program `mysys/thr_lock` to work. This ensures that your thread installation has even a remote chance to work!

Index

Q

R

S

T

U