```
-- This fails because audit_log_include_accounts is not NULL
SET GLOBAL audit_log_exclude_accounts = value;

-- To set audit_log_exclude_accounts, first set
-- audit_log_include_accounts to NULL
SET GLOBAL audit_log_include_accounts = NULL;
SET GLOBAL audit_log_exclude_accounts = value;
```

If you inspect the value of either variable, be aware that `SHOW VARIABLES` displays `NULL` as an empty string. To display `NULL` as `NULL`, use `SELECT` instead:

```
mysql> SHOW VARIABLES LIKE 'audit_log_include_accounts';
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| audit_log_include_accounts |       |
+----------------------------+-------+
mysql> SELECT @@audit_log_include_accounts;
+-----------------------------+
| @@audit_log_include_accounts |
+-----------------------------+
| NULL                        |
+-----------------------------+
```

If a user name or host name requires quoting because it contains a comma, space, or other special character, quote it using single quotes. If the variable value itself is quoted with single quotes, double each inner single quote or escape it with a backslash. The following statements each enable audit logging for the local `root` account and are equivalent, even though the quoting styles differ:

```
SET GLOBAL audit_log_include_accounts = 'root@localhost';
SET GLOBAL audit_log_include_accounts = '''root''@''localhost''';
SET GLOBAL audit_log_include_accounts = '\'root\'@\'localhost\'';
SET GLOBAL audit_log_include_accounts = "'root'@'localhost'";
```

The last statement does not work if the `ANSI_QUOTES` SQL mode is enabled because in that mode double quotes signify identifier quoting, not string quoting.

Legacy Event Filtering by Status

To filter audited events based on status, set the following system variables at server startup or runtime. These variables apply only for legacy audit log filtering. For JSON audit log filtering, different status variables apply; see Audit Log Options and Variables.

- `audit_log_connection_policy`: Logging policy for connection events
- `audit_log_statement_policy`: Logging policy for statement events

Each variable takes a value of `ALL` (log all associated events; this is the default), `ERRORS` (log only failed events), or `NONE` (do not log events). For example, to log all statement events but only failed connection events, use these settings:

```
SET GLOBAL audit_log_statement_policy = ALL;
SET GLOBAL audit_log_connection_policy = ERRORS;
```

Another policy system variable, `audit_log_policy`, is available but does not afford as much control as `audit_log_connection_policy` and `audit_log_statement_policy`. It can be set only at server startup. At runtime, it is a read-only variable. It takes a value of `ALL` (log all events; this is the default), `LOGINS` (log connection events), `QUERIES` (log statement events), or `NONE` (do not log events). For any of those values, the audit log plugin logs all selected events without distinction as to success or failure. Use of `audit_log_policy` at startup works as follows:

- If you do not set `audit_log_policy` or set it to its default of `ALL`, any explicit settings for `audit_log_connection_policy` or `audit_log_statement_policy` apply as specified. If not specified, they default to `ALL`.

- If you set `audit_log_policy` to a non-`ALL` value, that value takes precedence over and is used to set `audit_log_connection_policy` and `audit_log_statement_policy`, as indicated in the following table. If you also set either of those variables to a value other than their default of `ALL`, the server writes a message to the error log to indicate that their values are being overridden.

Startup audit_log_policy Value	Resulting audit_log_connection_policy Value	Resulting audit_log_statement_policy Value
`LOGINS`	`ALL`	`NONE`
`QUERIES`	`NONE`	`ALL`
`NONE`	`NONE`	`NONE`

6.4.5.11 Audit Log Reference

The following sections provide a reference to MySQL Enterprise Audit elements:

- Audit Log Tables
- Audit Log Functions
- Audit Log Option and Variable Reference
- Audit Log Options and Variables
- Audit Log Status Variables

To install the audit log tables and functions, use the instructions provided in Section 6.4.5.2, “Installing or Uninstalling MySQL Enterprise Audit”. Unless those objects are installed, the `audit_log` plugin operates in legacy mode. See Section 6.4.5.10, “Legacy Mode Audit Log Filtering”.

Audit Log Tables

MySQL Enterprise Audit uses tables in the `mysql` system database for persistent storage of filter and user account data. The tables can be accessed only by users who have privileges for that database. The tables use the `InnoDB` storage engine.

If these tables are missing, the `audit_log` plugin operates in legacy mode. See Section 6.4.5.10, “Legacy Mode Audit Log Filtering”.

The `audit_log_filter` table stores filter definitions. The table has these columns:

- `NAME`

  The filter name.

- `FILTER`

  The filter definition associated with the filter name. Definitions are stored as `JSON` values.

The `audit_log_user` table stores user account information. The table has these columns:

- `USER`

  The user name part of an account. For an account `user1@localhost`, the `USER` part is `user1`.

- `HOST`

  The host name part of an account. For an account `user1@localhost`, the `HOST` part is `localhost`.

- `FILTERNAME`

The name of the filter assigned to the account. The filter name associates the account with a filter defined in the `audit_log_filter` table.

Audit Log Functions

This section describes, for each audit log function, its purpose, calling sequence, and return value. For information about the conditions under which these functions can be invoked, see Section 6.4.5.7, “Audit Log Filtering”.

Each audit log function returns a string that indicates whether the operation succeeded. `OK` indicates success. `ERROR: message` indicates failure.

As of MySQL 8.0.19, audit log functions convert string arguments to `utf8mb4` and string return values are `utf8mb4` strings. Prior to MySQL 8.0.19, audit log functions treat string arguments as binary strings (which means they do not distinguish lettercase), and string return values are binary strings.

If an audit log function is invoked from within the `mysql` client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

These audit log functions are available:

- `audit_log_encryption_password_get([keyring_id])`

  This function fetches an audit log encryption password from the MySQL keyring, which must be enabled or an error occurs. Any keyring component or plugin can be used; for instructions, see Section 6.4.4, “The MySQL Keyring”.

  With no argument, the function retrieves the current encryption password as a binary string. An argument may be given to specify which audit log encryption password to retrieve. The argument must be the keyring ID of the current password or an archived password.

  For additional information about audit log encryption, see Encrypting Audit Log Files.

  Arguments:

  `keyring_id`: As of MySQL 8.0.17, this optional argument indicates the keyring ID of the password to retrieve. The maximum permitted length is 766 bytes. If omitted, the function retrieves the current password.

  Prior to MySQL 8.0.17, no argument is permitted. The function always retrieves the current password.

  Return value:

  The password string for success (up to 766 bytes), or `NULL` and an error for failure.

  Example:

  Retrieve the current password:

```
mysql> SELECT audit_log_encryption_password_get();
+-----------------------------------+
| audit_log_encryption_password_get() |
+-----------------------------------+
| secret                            |
+-----------------------------------+
```

  To retrieve a password by ID, you can determine which audit log keyring IDs exist by querying the Performance Schema `keyring_keys` table:

```
mysql> SELECT KEY_ID FROM performance_schema.keyring_keys
```

```
      WHERE KEY_ID LIKE 'audit_log%'
      ORDER BY KEY_ID;
+----------------------------+
| KEY_ID                     |
+----------------------------+
| audit_log-20190415T152248-1 |
| audit_log-20190415T153507-1 |
| audit_log-20190416T125122-1 |
| audit_log-20190416T141608-1 |
+----------------------------+
mysql> SELECT audit_log_encryption_password_get('audit_log-20190416T125122-1');
+-------------------------------------------------------------------+
| audit_log_encryption_password_get('audit_log-20190416T125122-1') |
+-------------------------------------------------------------------+
| segreto                                                           |
+-------------------------------------------------------------------+
```

- `audit_log_encryption_password_set(password)`

  Sets the current audit log encryption password to the argument and stores the password in the MySQL keyring. As of MySQL 8.0.19, the password is stored as a `utf8mb4` string. Prior to MySQL 8.0.19, the password is stored in binary form.

  If encryption is enabled, this function performs a log file rotation operation that renames the current log file, and begins a new log file encrypted with the password. The keyring must be enabled or an error occurs. Any keyring component or plugin can be used; for instructions, see Section 6.4.4, “The MySQL Keyring”.

  For additional information about audit log encryption, see Encrypting Audit Log Files.

  Arguments:

  `password`: The password string. The maximum permitted length is 766 bytes.

  Return value:

  1 for success, 0 for failure.

  Example:

```
mysql> SELECT audit_log_encryption_password_set(password);
+-------------------------------------------+
| audit_log_encryption_password_set(password) |
+-------------------------------------------+
| 1                                         |
+-------------------------------------------+
```

- `audit_log_filter_flush()`

  Calling any of the other filtering functions affects operational audit log filtering immediately and updates the audit log tables. If instead you modify the contents of those tables directly using statements such as `INSERT`, `UPDATE`, and `DELETE`, the changes do not affect filtering immediately. To flush your changes and make them operational, call `audit_log_filter_flush()`.

  Warning

  `audit_log_filter_flush()` should be used only after modifying the audit tables directly, to force reloading all filters. Otherwise, this function should be avoided. It is, in effect, a simplified version of unloading and reloading the `audit_log` plugin with `UNINSTALL PLUGIN` plus `INSTALL PLUGIN`.

  `audit_log_filter_flush()` affects all current sessions and detaches them from their previous filters. Current sessions are no longer logged unless they disconnect and reconnect, or execute a change-user operation.

If this function fails, an error message is returned and the audit log is disabled until the next successful call to `audit_log_filter_flush()`.

Arguments:

None.

Return value:

A string that indicates whether the operation succeeded. `OK` indicates success. `ERROR: message` indicates failure.

Example:

```
mysql> SELECT audit_log_filter_flush();
+--------------------------+
| audit_log_filter_flush() |
+--------------------------+
| OK                       |
+--------------------------+
```

- `audit_log_filter_remove_filter(filter_name)`

  Given a filter name, removes the filter from the current set of filters. It is not an error for the filter not to exist.

  If a removed filter is assigned to any user accounts, those users stop being filtered (they are removed from the `audit_log_user` table). Termination of filtering includes any current sessions for those users: They are detached from the filter and no longer logged.

  Arguments:

  - *`filter_name`*: A string that specifies the filter name.

  Return value:

  A string that indicates whether the operation succeeded. `OK` indicates success. `ERROR: message` indicates failure.

  Example:

```
mysql> SELECT audit_log_filter_remove_filter('SomeFilter');
+-------------------------------------------+
| audit_log_filter_remove_filter('SomeFilter') |
+-------------------------------------------+
| OK                                        |
+-------------------------------------------+
```

- `audit_log_filter_remove_user(user_name)`

  Given a user account name, cause the user to be no longer assigned to a filter. It is not an error if the user has no filter assigned. Filtering of current sessions for the user remains unaffected. New

connections for the user are filtered using the default account filter if there is one, and are not logged otherwise.

If the name is `%`, the function removes the default account filter that is used for any user account that has no explicitly assigned filter.

Arguments:

- *`user_name`*: The user account name as a string in *`user_name@host_name`* format, or `%` to represent the default account.

Return value:

A string that indicates whether the operation succeeded. `OK` indicates success. `ERROR: `*`message`* indicates failure.

Example:

```
mysql> SELECT audit_log_filter_remove_user('user1@localhost');
+-------------------------------------------------+
| audit_log_filter_remove_user('user1@localhost') |
+-------------------------------------------------+
| OK                                              |
+-------------------------------------------------+
```

- `audit_log_filter_set_filter(`*`filter_name, definition`*`)`

Given a filter name and definition, adds the filter to the current set of filters. If the filter already exists and is used by any current sessions, those sessions are detached from the filter and are no longer logged. This occurs because the new filter definition has a new filter ID that differs from its previous ID.

Arguments:

- *`filter_name`*: A string that specifies the filter name.
- *`definition`*: A `JSON` value that specifies the filter definition.

Return value:

A string that indicates whether the operation succeeded. `OK` indicates success. `ERROR: `*`message`* indicates failure.

Example:

```
mysql> SET @f = '{ "filter": { "log": false } }';
mysql> SELECT audit_log_filter_set_filter('SomeFilter', @f);
+-----------------------------------------------+
| audit_log_filter_set_filter('SomeFilter', @f) |
+-----------------------------------------------+
| OK                                            |
+-----------------------------------------------+
```

- `audit_log_filter_set_user(user_name, filter_name)`

  Given a user account name and a filter name, assigns the filter to the user. A user can be assigned only one filter, so if the user was already assigned a filter, the assignment is replaced. Filtering of current sessions for the user remains unaffected. New connections are filtered using the new filter.

  As a special case, the name `%` represents the default account. The filter is used for connections from any user account that has no explicitly assigned filter.

  Arguments:

  - *`user_name`*: The user account name as a string in *`user_name@host_name`* format, or `%` to represent the default account.
  - *`filter_name`*: A string that specifies the filter name.

  Return value:

  A string that indicates whether the operation succeeded. `OK` indicates success. `ERROR: message` indicates failure.

  Example:

```
mysql> SELECT audit_log_filter_set_user('user1@localhost', 'SomeFilter');
+-----------------------------------------------------------+
| audit_log_filter_set_user('user1@localhost', 'SomeFilter') |
+-----------------------------------------------------------+
| OK                                                        |
+-----------------------------------------------------------+
```

- `audit_log_read([arg])`

  Reads the audit log and returns a `JSON` string result. If the audit log format is not `JSON`, an error occurs.

  With no argument or a `JSON` hash argument, `audit_log_read()` reads events from the audit log and returns a `JSON` string containing an array of audit events. Items in the hash argument influence how reading occurs, as described later. Each element in the returned array is an event represented as a `JSON` hash, with the exception that the last element may be a `JSON null` value to indicate no following events are available to read.

  With an argument consisting of a `JSON null` value, `audit_log_read()` closes the current read sequence.

  For additional details about the audit log-reading process, see Section 6.4.5.6, “Reading Audit Log Files”.

  Arguments:

  To obtain a bookmark for the most recently written event, call `audit_log_read_bookmark()`.

  `arg`: The argument is optional. If omitted, the function reads events from the current position. If present, the argument can be a `JSON null` value to close the read sequence, or a `JSON` hash. Within a hash argument, items are optional and control aspects of the read operation such as the

position at which to begin reading or how many events to read. The following items are significant (other items are ignored):

- `start`: The position within the audit log of the first event to read. The position is given as a timestamp and the read starts from the first event that occurs on or after the timestamp value. The `start` item has this format, where *`value`* is a literal timestamp value:

  ```
  "start": { "timestamp": "value" }
  ```

  The `start` item is permitted as of MySQL 8.0.22.

- `timestamp`, `id`: The position within the audit log of the first event to read. The `timestamp` and `id` items together comprise a bookmark that uniquely identify a particular event. If an `audit_log_read()` argument includes either item, it must include both to completely specify a position or an error occurs.

- `max_array_length`: The maximum number of events to read from the log. If this item is omitted, the default is to read to the end of the log or until the read buffer is full, whichever comes first.

To specify a starting position to `audit_log_read()`, pass a hash argument that includes either a `start` item or a bookmark consisting of `timestamp` and `id` items. If a hash argument includes both a `start` item and a bookmark, an error occurs.

If a hash argument specifies no starting position, reading continues from the current position.

If a timestamp value includes no time part, a time part of `00:00:00` is assumed.

Return value:

If the call succeeds, the return value is a `JSON` string containing an array of audit events, or a `JSON` `null` value if that was passed as the argument to close the read sequence. If the call fails, the return value is `NULL` and an error occurs.

Example:

```
mysql> SELECT audit_log_read(audit_log_read_bookmark());
+------------------------------------------------------------------------+
| audit_log_read(audit_log_read_bookmark())                              |
+------------------------------------------------------------------------+
| [ {"timestamp":"2020-05-18 22:41:24","id":0,"class":"connection", ... |
+------------------------------------------------------------------------+
mysql> SELECT audit_log_read('null');
+------------------------+
| audit_log_read('null') |
+------------------------+
| null                   |
+------------------------+
```

Notes:

Prior to MySQL 8.0.19, string return values are binary `JSON` strings. For information about converting such values to nonbinary strings, see Section 6.4.5.6, “Reading Audit Log Files”.

- `audit_log_read_bookmark()`

  Returns a `JSON` string representing a bookmark for the most recently written audit log event. If the audit log format is not `JSON`, an error occurs.

  The bookmark is a `JSON` hash with `timestamp` and `id` items that uniquely identify the position of an event within the audit log. It is suitable for passing to `audit_log_read()` to indicate to that function the position at which to begin reading.

  For additional details about the audit log-reading process, see Section 6.4.5.6, “Reading Audit Log Files”.

  Arguments:

  None.

  Return value:

  A `JSON` string containing a bookmark for success, or `NULL` and an error for failure.

  Example:

```
mysql> SELECT audit_log_read_bookmark();
+------------------------------------------------+
| audit_log_read_bookmark()                      |
+------------------------------------------------+
| { "timestamp": "2019-10-03 21:03:44", "id": 0 } |
+------------------------------------------------+
```

  Notes:

  Prior to MySQL 8.0.19, string return values are binary `JSON` strings. For information about converting such values to nonbinary strings, see Section 6.4.5.6, “Reading Audit Log Files”.

- `audit_log_rotate()`

  Arguments:

  None.

  Return value:

  The renamed file name.

  Example:

```
mysql> SELECT audit_log_rotate();
```

  Using `audit_log_rotate()` requires the `AUDIT_ADMIN` privilege.

Audit Log Option and Variable Reference

Table 6.44 Audit Log Option and Variable Reference

Name	Cmd-Line	Option File	System Var	Status Var	Var Scope	Dynamic
audit-log	Yes	Yes				
audit_log_buffer_size	Yes	Yes	Yes		Global	No
audit_log_compression	Yes	Yes	Yes		Global	No
audit_log_connection_policy	Yes	Yes	Yes		Global	Yes
audit_log_current_session			Yes		Both	No
Audit_log_current_size				Yes	Global	No

Name	Cmd-Line	Option File	System Var	Status Var	Var Scope	Dynamic
audit_log_disable	Yes	Yes	Yes		Global	Yes
audit_log_encryption	Yes	Yes	Yes		Global	No
Audit_log_event_max_drop_size				Yes	Global	No
Audit_log_events				Yes	Global	No
Audit_log_events_filtered				Yes	Global	No
Audit_log_events_lost				Yes	Global	No
Audit_log_events_written				Yes	Global	No
audit_log_exclude_accounts	Yes	Yes	Yes		Global	Yes
audit_log_file	Yes	Yes	Yes		Global	No
audit_log_filter_id			Yes		Both	No
audit_log_flush			Yes		Global	Yes
audit_log_format	Yes	Yes	Yes		Global	No
audit_log_include_accounts	Yes	Yes	Yes		Global	Yes
audit_log_max_size	Yes	Yes	Yes		Global	Yes
audit_log_password_history_keep_days	Yes	Yes	Yes		Global	Yes
audit_log_policy	Yes	Yes	Yes		Global	No
audit_log_prune_seconds	Yes	Yes	Yes		Global	Yes
audit_log_read_buffer_size	Yes	Yes	Yes		Varies	Varies
audit_log_rotate_on_size	Yes	Yes	Yes		Global	Yes
audit_log_statement_policy	Yes	Yes	Yes		Global	Yes
audit_log_strategy	Yes	Yes	Yes		Global	No
Audit_log_total_size				Yes	Global	No
Audit_log_write_waits				Yes	Global	No

Audit Log Options and Variables

This section describes the command options and system variables that configure operation of MySQL Enterprise Audit. If values specified at startup time are incorrect, the `audit_log` plugin may fail to initialize properly and the server does not load it. In this case, the server may also produce error messages for other audit log settings because it does not recognize them.

To configure activation of the audit log plugin, use this option:

- `--audit-log[=value]`

Command-Line Format	`--audit-log[=value]`
Type	Enumeration
Default Value	`ON`

Valid Values	`ON`
	`OFF`
	`FORCE`
	`FORCE_PLUS_PERMANENT`

This option controls how the server loads the `audit_log` plugin at startup. It is available only if the plugin has been previously registered with `INSTALL PLUGIN` or is loaded with `--plugin-load` or `--plugin-load-add`. See Section 6.4.5.2, “Installing or Uninstalling MySQL Enterprise Audit”.

The option value should be one of those available for plugin-loading options, as described in Section 5.6.1, “Installing and Uninstalling Plugins”. For example, `--audit-log=FORCE_PLUS_PERMANENT` tells the server to load the plugin and prevent it from being removed while the server is running.

If the audit log plugin is enabled, it exposes several system variables that permit control over logging:

```
mysql> SHOW VARIABLES LIKE 'audit_log%';
+--------------------------------------+--------------+
| Variable_name                        | Value        |
+--------------------------------------+--------------+
| audit_log_buffer_size                | 1048576      |
| audit_log_compression                | NONE         |
| audit_log_connection_policy          | ALL          |
| audit_log_current_session            | OFF          |
| audit_log_disable                    | OFF          |
| audit_log_encryption                 | NONE         |
| audit_log_exclude_accounts           |              |
| audit_log_file                       | audit.log    |
| audit_log_filter_id                  | 0            |
| audit_log_flush                      | OFF          |
| audit_log_format                     | NEW          |
| audit_log_format_unix_timestamp      | OFF          |
| audit_log_include_accounts           |              |
| audit_log_max_size                   | 0            |
| audit_log_password_history_keep_days | 0            |
| audit_log_policy                     | ALL          |
| audit_log_prune_seconds              | 0            |
| audit_log_read_buffer_size           | 32768        |
| audit_log_rotate_on_size             | 0            |
| audit_log_statement_policy           | ALL          |
| audit_log_strategy                   | ASYNCHRONOUS |
+--------------------------------------+--------------+
```

You can set any of these variables at server startup, and some of them at runtime. Those that are available only for legacy mode audit log filtering are so noted.

- `audit_log_buffer_size`

Command-Line Format	`--audit-log-buffer-size=#`
System Variable	`audit_log_buffer_size`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`1048576`
Minimum Value	`4096`
Maximum Value (64-bit platforms)	`18446744073709547520`
Maximum Value (32-bit platforms)	`4294967295`
Unit	bytes
Block Size	`4096`

When the audit log plugin writes events to the log asynchronously, it uses a buffer to store event contents prior to writing them. This variable controls the size of that buffer, in bytes. The server adjusts the value to a multiple of 4096. The plugin uses a single buffer, which it allocates when

it initializes and removes when it terminates. The plugin allocates this buffer only if logging is asynchronous.

- `audit_log_compression`

Command-Line Format	`--audit-log-compression=value`
System Variable	`audit_log_compression`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`NONE`
Valid Values	`NONE` `GZIP`

The type of compression for the audit log file. Permitted values are `NONE` (no compression; the default) and `GZIP` (GNU Zip compression). For more information, see Compressing Audit Log Files.

- `audit_log_connection_policy`

Command-Line Format	`--audit-log-connection-policy=value`
System Variable	`audit_log_connection_policy`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`ALL`

Valid Values	`ALL`
	`ERRORS`
	`NONE`

Note

This variable applies only to legacy mode audit log filtering (see Section 6.4.5.10, “Legacy Mode Audit Log Filtering”).

The policy controlling how the audit log plugin writes connection events to its log file. The following table shows the permitted values.

Value	Description
`ALL`	Log all connection events
`ERRORS`	Log only failed connection events
`NONE`	Do not log connection events

Note

At server startup, any explicit value given for `audit_log_connection_policy` may be overridden if `audit_log_policy` is also specified, as described in Section 6.4.5.5, “Configuring Audit Logging Characteristics”.

- `audit_log_current_session`

System Variable	`audit_log_current_session`
Scope	Global, Session
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`depends on filtering policy`

Whether audit logging is enabled for the current session. The session value of this variable is read only. It is set when the session begins based on the values of the `audit_log_include_accounts` and `audit_log_exclude_accounts` system variables. The audit log plugin uses the session value to determine whether to audit events for the session. (There is a global value, but the plugin does not use it.)

- `audit_log_disable`

Command-Line Format	`--audit-log-disable[={OFF|ON}]`
Introduced	8.0.28
System Variable	`audit_log_disable`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`OFF`

Permits disabling audit logging for all connecting and connected sessions. In addition to the `SYSTEM_VARIABLES_ADMIN` privilege, disabling audit logging requires the `AUDIT_ADMIN` privilege. See Section 6.4.5.9, “Disabling Audit Logging”.

- `audit_log_encryption`

Command-Line Format	`--audit-log-encryption=value`
System Variable	`audit_log_encryption`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`NONE`
Valid Values	`NONE` `AES`

The type of encryption for the audit log file. Permitted values are `NONE` (no encryption; the default) and `AES` (AES-256-CBC cipher encryption). For more information, see Encrypting Audit Log Files.

- `audit_log_exclude_accounts`

Command-Line Format	`--audit-log-exclude-accounts=value`
System Variable	`audit_log_exclude_accounts`
Scope	Global

Dynamic	Yes
SET_VAR Hint Applies	No
Type	String
Default Value	NULL

Note

This variable applies only to legacy mode audit log filtering (see Section 6.4.5.10, “Legacy Mode Audit Log Filtering”).

The accounts for which events should not be logged. The value should be NULL or a string containing a list of one or more comma-separated account names. For more information, see Section 6.4.5.7, “Audit Log Filtering”.

Modifications to audit_log_exclude_accounts affect only connections created subsequent to the modification, not existing connections.

- audit_log_file

Command-Line Format	--audit-log-file=file_name
System Variable	audit_log_file
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	File name
Default Value	audit.log

The base name and suffix of the file to which the audit log plugin writes events. The default value is audit.log, regardless of logging format. To have the name suffix correspond to the format, set the name explicitly, choosing a different suffix (for example, audit.xml for XML format, audit.json for JSON format).

If the value of audit_log_file is a relative path name, the plugin interprets it relative to the data directory. If the value is a full path name, the plugin uses the value as is. A full path name may be useful if it is desirable to locate audit files on a separate file system or directory. For security reasons, write the audit log file to a directory accessible only to the MySQL server and to users with a legitimate reason to view the log.

For details about how the audit log plugin interprets the audit_log_file value and the rules for file renaming that occurs at plugin initialization and termination, see Naming Conventions for Audit Log Files.

The audit log plugin uses the directory containing the audit log file (determined from the audit_log_file value) as the location to search for readable audit log files. From these log files and the current file, the plugin constructs a list of the ones that are subject to use with the audit log bookmarking and reading functions. See Section 6.4.5.6, “Reading Audit Log Files”.

- audit_log_filter_id

System Variable	audit_log_filter_id
Scope	Global, Session
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer

Default Value	1
Minimum Value	0
Maximum Value	4294967295

The session value of this variable indicates the internally maintained ID of the audit filter for the current session. A value of 0 means that the session has no filter assigned.

- audit_log_flush

System Variable	audit_log_flush
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	OFF

Note

The audit_log_flush variable is deprecated as of MySQL 8.0.31; expect support for it to be removed in a future version of MySQL. It is superseded by the audit_log_rotate() function.

If audit_log_rotate_on_size is 0, automatic audit log file rotation is disabled and rotation occurs only when performed manually. In that case, enabling audit_log_flush by setting it to 1 or ON causes the audit log plugin to close and reopen its log file to flush it. (The variable value remains OFF so that you need not disable it explicitly before enabling it again to perform another flush.) For more information, see Section 6.4.5.5, “Configuring Audit Logging Characteristics”.

- audit_log_format

Command-Line Format	--audit-log-format=value
System Variable	audit_log_format
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Enumeration
Default Value	NEW
Valid Values	OLD NEW JSON

The audit log file format. Permitted values are OLD (old-style XML), NEW (new-style XML; the default), and JSON. For details about each format, see Section 6.4.5.4, “Audit Log File Formats”.

- audit_log_format_unix_timestamp

Command-Line Format	--audit-log-format-unix-timestamp[={OFF|ON}]
Introduced	8.0.26
System Variable	audit_log_format_unix_timestamp

Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	OFF

This variable applies only for JSON-format audit log output. When that is true, enabling this variable causes each log file record to include a `time` field. The field value is an integer that represents the UNIX timestamp value indicating the date and time when the audit event was generated.

Changing the value of this variable at runtime causes log file rotation so that, for a given JSON-format log file, all records in the file either do or do not include the `time` field.

Setting the runtime value of `audit_log_format_unix_timestamp` requires the `AUDIT_ADMIN` privilege, in addition to the `SYSTEM_VARIABLES_ADMIN` privilege (or the deprecated `SUPER` privilege) normally required to set a global system variable runtime value.

- `audit_log_include_accounts`

Command-Line Format	`--audit-log-include-accounts=value`
System Variable	`audit_log_include_accounts`
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	String
Default Value	NULL

Note

This variable applies only to legacy mode audit log filtering (see Section 6.4.5.10, “Legacy Mode Audit Log Filtering”).

The accounts for which events should be logged. The value should be `NULL` or a string containing a list of one or more comma-separated account names. For more information, see Section 6.4.5.7, “Audit Log Filtering”.

Modifications to `audit_log_include_accounts` affect only connections created subsequent to the modification, not existing connections.

- `audit_log_max_size`

Command-Line Format	`--audit-log-max-size=#`
Introduced	8.0.26
System Variable	`audit_log_max_size`
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	0
Minimum Value	0
Maximum Value (Windows)	4294967295

Maximum Value (Other)	18446744073709551615
Unit	bytes
Block Size	4096

audit_log_max_size pertains to audit log file pruning, which is supported for JSON-format log files only. It controls pruning based on combined log file size:

- A value of 0 (the default) disables size-based pruning. No size limit is enforced.
- A value greater than 0 enables size-based pruning. The value is the combined size above which audit log files become subject to pruning.

If you set audit_log_max_size to a value that is not a multiple of 4096, it is truncated to the nearest multiple. In particular, setting it to a value less than 4096 sets it to 0 and no size-based pruning occurs.

If both audit_log_max_size and audit_log_rotate_on_size are greater than 0, audit_log_max_size should be more than 7 times the value of audit_log_rotate_on_size. Otherwise, a warning is written to the server error log because in this case the "granularity" of size-based pruning may be insufficient to prevent removal of all or most rotated log files each time it occurs.

Note

Setting audit_log_max_size by itself is not sufficient to cause log file pruning to occur because the pruning algorithm uses audit_log_rotate_on_size, audit_log_max_size, and audit_log_prune_seconds in conjunction. For details, see Space Management of Audit Log Files.

- audit_log_password_history_keep_days

Command-Line Format	--audit-log-password-history-keep-days=#
Introduced	8.0.17
System Variable	audit_log_password_history_keep_days
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	0
Minimum Value	0
Maximum Value	4294967295
Unit	days

The audit log plugin implements log file encryption using encryption passwords stored in the MySQL keyring (see Encrypting Audit Log Files). The plugin also implements password history, which includes password archiving and expiration (removal).

When the audit log plugin creates a new encryption password, it archives the previous password, if one exists, for later use. The audit_log_password_history_keep_days variable controls automatic removal of expired archived passwords. Its value indicates the number of days after which

archived audit log encryption passwords are removed. The default of 0 disables password expiration: the password retention period is forever.

New audit log encryption passwords are created under these circumstances:

- During plugin initialization, if the plugin finds that log file encryption is enabled, it checks whether the keyring contains an audit log encryption password. If not, the plugin automatically generates a random initial encryption password.
- When the `audit_log_encryption_password_set()` function is called to set a specific password.

In each case, the plugin stores the new password in the key ring and uses it to encrypt new log files.

Removal of expired audit log encryption passwords occurs under these circumstances:

- During plugin initialization.
- When the `audit_log_encryption_password_set()` function is called.
- When the runtime value of `audit_log_password_history_keep_days` is changed from its current value to a value greater than 0. Runtime value changes occur for `SET` statements that use the `GLOBAL` or `PERSIST` keyword, but not the `PERSIST_ONLY` keyword. `PERSIST_ONLY` writes the variable setting to `mysqld-auto.cnf`, but has no effect on the runtime value.

When password removal occurs, the current value of `audit_log_password_history_keep_days` determines which passwords to remove:

- If the value is 0, the plugin removes no passwords.
- If the value is *N* > 0, the plugin removes passwords more than *N* days old.

Note

Take care not to expire old passwords that are still needed to read archived encrypted log files.

If you normally leave password expiration disabled (that is, `audit_log_password_history_keep_days` has a value of 0), it is possible to perform an on-demand cleanup operation by temporarily assigning the variable a value greater than zero. For example, to expire passwords older than 365 days, do this:

```
SET GLOBAL audit_log_password_history_keep_days = 365;
SET GLOBAL audit_log_password_history_keep_days = 0;
```

Setting the runtime value of `audit_log_password_history_keep_days` requires the `AUDIT_ADMIN` privilege, in addition to the `SYSTEM_VARIABLES_ADMIN` privilege (or the deprecated `SUPER` privilege) normally required to set a global system variable runtime value.

- `audit_log_policy`

Command-Line Format	`--audit-log-policy=value`
System Variable	`audit_log_policy`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`ALL`

Valid Values	`ALL`
	`LOGINS`
	`QUERIES`
	`NONE`

Note

This variable applies only to legacy mode audit log filtering (see Section 6.4.5.10, “Legacy Mode Audit Log Filtering”).

The policy controlling how the audit log plugin writes events to its log file. The following table shows the permitted values.

Value	Description
`ALL`	Log all events
`LOGINS`	Log only login events
`QUERIES`	Log only query events
`NONE`	Log nothing (disable the audit stream)

`audit_log_policy` can be set only at server startup. At runtime, it is a read-only variable. Two other system variables, `audit_log_connection_policy` and `audit_log_statement_policy`, provide finer control over logging policy and can be set either at startup or at runtime. If you use `audit_log_policy` at startup instead of the other two variables, the server uses its value to set those variables. For more information about the policy variables and their interaction, see Section 6.4.5.5, “Configuring Audit Logging Characteristics”.

- `audit_log_prune_seconds`

Command-Line Format	`--audit-log-prune-seconds=#`
Introduced	8.0.24
System Variable	`audit_log_prune_seconds`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`0`
Minimum Value	`0`
Maximum Value (Windows)	`4294967295`
Maximum Value (Other)	`18446744073709551615`
Unit	bytes

`audit_log_prune_seconds` pertains to audit log file pruning, which is supported for JSON-format log files only. It controls pruning based on log file age:

- A value of 0 (the default) disables age-based pruning. No age limit is enforced.

- A value greater than 0 enables age-based pruning. The value is the number of seconds after which audit log files become subject to pruning.

Note

Setting `audit_log_prune_seconds` by itself is not sufficient to cause log file pruning to occur because the pruning algorithm uses `audit_log_rotate_on_size`, `audit_log_max_size`, and `audit_log_prune_seconds` in conjunction. For details, see Space Management of Audit Log Files.

- `audit_log_read_buffer_size`

Command-Line Format	`--audit-log-read-buffer-size=#`
System Variable	`audit_log_read_buffer_size`
Scope (≥ 8.0.12)	Global, Session
Scope (8.0.11)	Global
Dynamic (≥ 8.0.12)	Yes
Dynamic (8.0.11)	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value (≥ 8.0.12)	`32768`
Default Value (8.0.11)	`1048576`
Minimum Value (≥ 8.0.12)	`32768`
Minimum Value (8.0.11)	`1024`
Maximum Value	`4194304`
Unit	bytes

The buffer size for reading from the audit log file, in bytes. The `audit_log_read()` function reads no more than this many bytes. Log file reading is supported only for JSON log format. For more information, see Section 6.4.5.6, “Reading Audit Log Files”.

As of MySQL 8.0.12, this variable has a default of 32KB and can be set at runtime. Each client should set its session value of `audit_log_read_buffer_size` appropriately for its use of `audit_log_read()`. Prior to MySQL 8.0.12, `audit_log_read_buffer_size` has a default of 1MB, affects all clients, and can be changed only at server startup.

- `audit_log_rotate_on_size`

Command-Line Format	`--audit-log-rotate-on-size=#`
System Variable	`audit_log_rotate_on_size`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`0`
Minimum Value	`0`
Maximum Value	`18446744073709551615`
Unit	bytes

Block Size	4096

If audit_log_rotate_on_size is 0, the audit log plugin does not perform automatic size-based log file rotation. If rotation is to occur, you must perform it manually; see Manual Audit Log File Rotation (Before MySQL 8.0.31).

If audit_log_rotate_on_size is greater than 0, automatic size-based log file rotation occurs. Whenever a write to the log file causes its size to exceed the audit_log_rotate_on_size value, the audit log plugin renames the current log file and opens a new current log file using the original name.

If you set audit_log_rotate_on_size to a value that is not a multiple of 4096, it is truncated to the nearest multiple. In particular, setting it to a value less than 4096 sets it to 0 and no rotation occurs, except manually.

Note

audit_log_rotate_on_size controls whether audit log file rotation occurs. It can also be used in conjunction with audit_log_max_size and audit_log_prune_seconds to configure pruning of rotated JSON-format log files. For details, see Space Management of Audit Log Files.

- audit_log_statement_policy

Command-Line Format	--audit-log-statement-policy=value
System Variable	audit_log_statement_policy
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Enumeration
Default Value	ALL
Valid Values	ALL ERRORS NONE

Note

This variable applies only to legacy mode audit log filtering (see Section 6.4.5.10, “Legacy Mode Audit Log Filtering”).

The policy controlling how the audit log plugin writes statement events to its log file. The following table shows the permitted values.

Value	Description
ALL	Log all statement events
ERRORS	Log only failed statement events
NONE	Do not log statement events

Note

At server startup, any explicit value given for audit_log_statement_policy may be overridden if

> `audit_log_policy` is also specified, as described in Section 6.4.5.5, “Configuring Audit Logging Characteristics”.

- `audit_log_strategy`

Command-Line Format	`--audit-log-strategy=value`
System Variable	`audit_log_strategy`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`ASYNCHRONOUS`

Valid Values	`ASYNCHRONOUS`
	`PERFORMANCE`
	`SEMISYNCHRONOUS`
	`SYNCHRONOUS`

The logging method used by the audit log plugin. These strategy values are permitted:

- `ASYNCHRONOUS`: Log asynchronously. Wait for space in the output buffer.
- `PERFORMANCE`: Log asynchronously. Drop requests for which there is insufficient space in the output buffer.
- `SEMISYNCHRONOUS`: Log synchronously. Permit caching by the operating system.
- `SYNCHRONOUS`: Log synchronously. Call `sync()` after each request.

Audit Log Status Variables

If the audit log plugin is enabled, it exposes several status variables that provide operational information. These variables are available for legacy mode audit filtering and JSON mode audit filtering.

- `Audit_log_current_size`

  The size of the current audit log file. The value increases when an event is written to the log and is reset to 0 when the log is rotated.

- `Audit_log_event_max_drop_size`

  The size of the largest dropped event in performance logging mode. For a description of logging modes, see Section 6.4.5.5, “Configuring Audit Logging Characteristics”.

- `Audit_log_events`

  The number of events handled by the audit log plugin, whether or not they were written to the log based on filtering policy (see Section 6.4.5.5, “Configuring Audit Logging Characteristics”).

- `Audit_log_events_filtered`

  The number of events handled by the audit log plugin that were filtered (not written to the log) based on filtering policy (see Section 6.4.5.5, “Configuring Audit Logging Characteristics”).

- `Audit_log_events_lost`

The number of events lost in performance logging mode because an event was larger than the available audit log buffer space. This value may be useful for assessing how to set `audit_log_buffer_size` to size the buffer for performance mode. For a description of logging modes, see Section 6.4.5.5, “Configuring Audit Logging Characteristics”.

- `Audit_log_events_written`

  The number of events written to the audit log.

- `Audit_log_total_size`

  The total size of events written to all audit log files. Unlike `Audit_log_current_size`, the value of `Audit_log_total_size` increases even when the log is rotated.

- `Audit_log_write_waits`

  The number of times an event had to wait for space in the audit log buffer in asynchronous logging mode. For a description of logging modes, see Section 6.4.5.5, “Configuring Audit Logging Characteristics”.

6.4.5.12 Audit Log Restrictions

MySQL Enterprise Audit is subject to these general restrictions:

- Only SQL statements are logged. Changes made by no-SQL APIs, such as memcached, Node.JS, and the NDB API, are not logged.
- Only top-level statements are logged, not statements within stored programs such as triggers or stored procedures.
- Contents of files referenced by statements such as `LOAD DATA` are not logged.

NDB Cluster. It is possible to use MySQL Enterprise Audit with MySQL NDB Cluster, subject to the following conditions:

- All changes to be logged must be done using the SQL interface. Changes using no-SQL interfaces, such as those provided by the NDB API, memcached, or ClusterJ, are not logged.
- The plugin must be installed on each MySQL server that is used to execute SQL on the cluster.
- Audit plugin data must be aggregated amongst all MySQL servers used with the cluster. This aggregation is the responsibility of the application or user.

6.4.6 The Audit Message Component

As of MySQL 8.0.14, the `audit_api_message_emit` component enables applications to add their own message events to the audit log, using the `audit_api_message_emit_udf()` function.

The `audit_api_message_emit` component cooperates with all plugins of audit type. For concreteness, examples use the `audit_log` plugin described in Section 6.4.5, “MySQL Enterprise Audit”.

- Installing or Uninstalling the Audit Message Component
- Audit Message Function

Installing or Uninstalling the Audit Message Component

To be usable by the server, the component library file must be located in the MySQL plugin directory (the directory named by the `plugin_dir` system variable). If necessary, configure the plugin directory location by setting the value of `plugin_dir` at server startup.

To install the `audit_api_message_emit` component, use this statement:

```
INSTALL COMPONENT "file://component_audit_api_message_emit";
```

Component installation is a one-time operation that need not be done per server startup. `INSTALL COMPONENT` loads the component, and also registers it in the `mysql.component` system table to cause it to be loaded during subsequent server startups.

To uninstall the `audit_api_message_emit` component, use this statement:

```
UNINSTALL COMPONENT "file://component_audit_api_message_emit";
```

`UNINSTALL COMPONENT` unloads the component, and unregisters it from the `mysql.component` system table to cause it not to be loaded during subsequent server startups.

Because installing and uninstalling the `audit_api_message_emit` component installs and uninstalls the `audit_api_message_emit_udf()` function that the component implements, it is not necessary to use `CREATE FUNCTION` or `DROP FUNCTION` to do so.

Audit Message Function

This section describes the `audit_api_message_emit_udf()` function implemented by the `audit_api_message_emit` component.

Before using the audit message function, install the audit message component according to the instructions provided at Installing or Uninstalling the Audit Message Component.

- `audit_api_message_emit_udf(`*`component`*`, `*`producer`*`, `*`message`*`[, `*`key`*`, `*`value`*`] ...)`

  Adds a message event to the audit log. Message events include component, producer, and message strings of the caller's choosing, and optionally a set of key-value pairs.

  An event posted by this function is sent to all enabled plugins of audit type, each of which handles the event according to its own rules. If no plugin of audit type is enabled, posting the event has no effect.

  Arguments:

  - *`component`*: A string that specifies a component name.
  - *`producer`*: A string that specifies a producer name.
  - *`message`*: A string that specifies the event message.
  - *`key`*, *`value`*: Events may include 0 or more key-value pairs that specify an arbitrary application-provided data map. Each *`key`* argument is a string that specifies a name for its immediately following *`value`* argument. Each *`value`* argument specifies a value for its immediately following *`key`* argument. Each *`value`* can be a string or numeric value, or `NULL`.

  Return value:

  The string `OK` to indicate success. An error occurs if the function fails.

  Example:

```
mysql> SELECT audit_api_message_emit_udf('component_text',
                                         'producer_text',
                                         'message_text',
                                         'key1', 'value1',
                                         'key2', 123,
                                         'key3', NULL) AS 'Message';
+---------+
| Message |
+---------+
| OK      |
+---------+
```

Additional information:

Each audit plugin that receives an event posted by `audit_api_message_emit_udf()` logs the event in plugin-specific format. For example, the `audit_log` plugin (see Section 6.4.5, “MySQL Enterprise Audit”) logs message values as follows, depending on the log format configured by the `audit_log_format` system variable:

- JSON format (`audit_log_format=JSON`):

```
{
  ...
  "class": "message",
  "event": "user",
  ...
  "message_data": {
    "component": "component_text",
    "producer": "producer_text",
    "message": "message_text",
    "map": {
      "key1": "value1",
      "key2": 123,
      "key3": null
    }
  }
}
```

- New-style XML format (`audit_log_format=NEW`):

```
<AUDIT_RECORD>
 ...
 <NAME>Message</NAME>
 ...
 <COMMAND_CLASS>user</COMMAND_CLASS>
 <COMPONENT>component_text</COMPONENT>
 <PRODUCER>producer_text</PRODUCER>
 <MESSAGE>message_text</MESSAGE>
 <MAP>
   <ELEMENT>
     <KEY>key1</KEY>
     <VALUE>value1</VALUE>
   </ELEMENT>
   <ELEMENT>
     <KEY>key2</KEY>
     <VALUE>123</VALUE>
   </ELEMENT>
   <ELEMENT>
     <KEY>key3</KEY>
     <VALUE/>
   </ELEMENT>
 </MAP>
</AUDIT_RECORD>
```

- Old-style XML format (`audit_log_format=OLD`):

```
<AUDIT_RECORD
  ...
  NAME="Message"
  ...
  COMMAND_CLASS="user"
  COMPONENT="component_text"
  PRODUCER="producer_text"
```

```
  MESSAGE="message_text"/>
```

Note

Message events logged in old-style XML format do not include the key-value map due to representational constraints imposed by this format.

Messages posted by `audit_api_message_emit_udf()` have an event class of `MYSQL_AUDIT_MESSAGE_CLASS` and a subclass of `MYSQL_AUDIT_MESSAGE_USER`. (Internally generated audit messages have the same class and a subclass of `MYSQL_AUDIT_MESSAGE_INTERNAL`; this subclass currently is unused.) To refer to such events in `audit_log` filtering rules, use a `class` element with a `name` value of `message`. For example:

```
{
  "filter": {
    "class": {
      "name": "message"
    }
  }
}
```

Should it be necessary to distinguish user-generated and internally generated message events, test the `subclass` value against `user` or `internal`.

Filtering based on the contents of the key-value map is not supported.

For information about writing filtering rules, see Section 6.4.5.7, “Audit Log Filtering”.

6.4.7 MySQL Enterprise Firewall

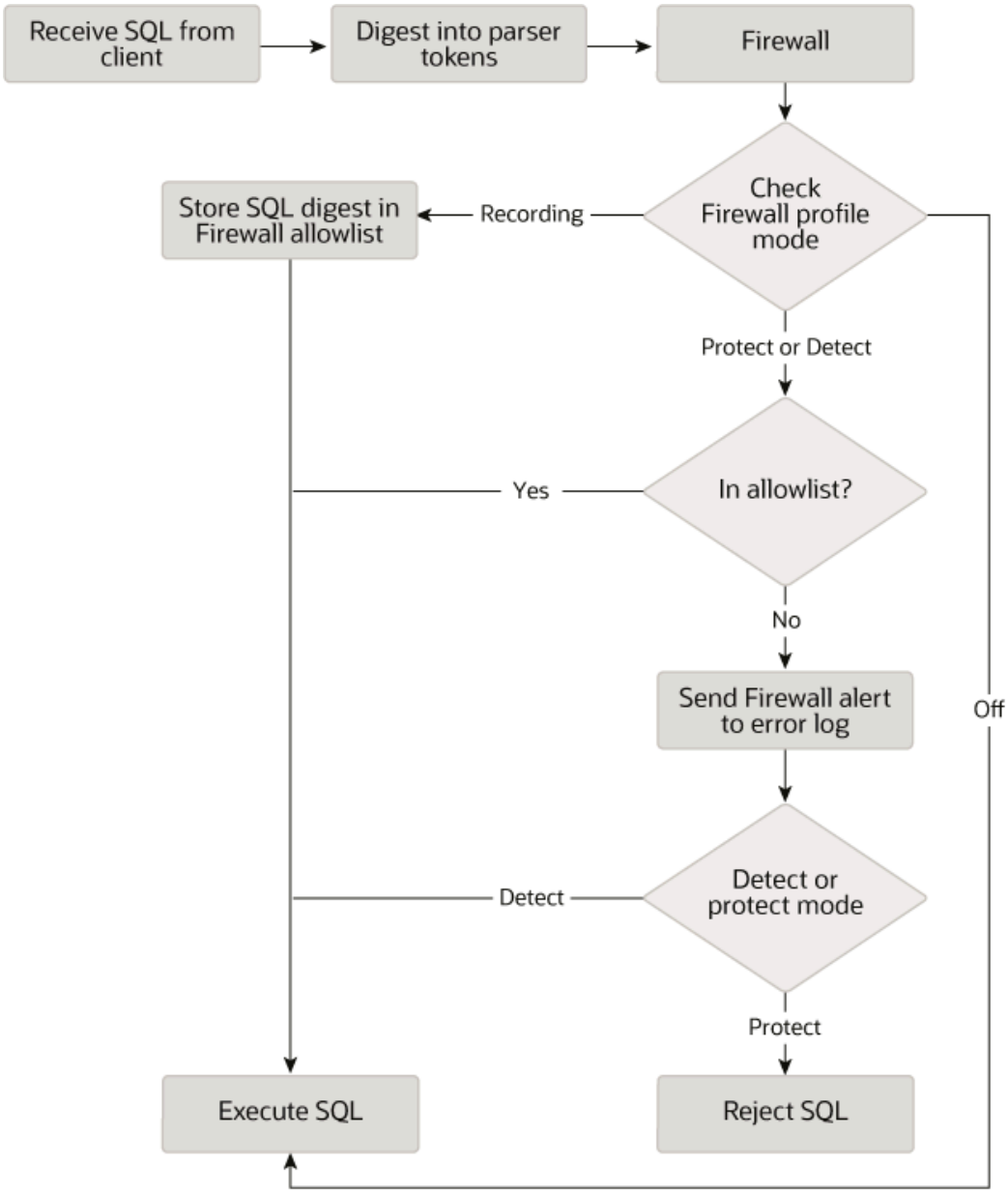
Note

MySQL Enterprise Firewall is an extension included in MySQL Enterprise Edition, a commercial product. To learn more about commercial products, see https://www.mysql.com/products/.

MySQL Enterprise Edition includes MySQL Enterprise Firewall, an application-level firewall that enables database administrators to permit or deny SQL statement execution based on matching against lists of accepted statement patterns. This helps harden MySQL Server against attacks such as SQL injection or attempts to exploit applications by using them outside of their legitimate query workload characteristics.

Each MySQL account registered with the firewall has its own statement allowlist, enabling protection to be tailored per account. For a given account, the firewall can operate in recording, protecting, or detecting mode, for training in the accepted statement patterns, active protection against unacceptable statements, or passive detection of unacceptable statements. The diagram illustrates how the firewall processes incoming statements in each mode.

Figure 6.1 MySQL Enterprise Firewall Operation



The following sections describe the elements of MySQL Enterprise Firewall, discuss how to install and use it, and provide reference information for its elements.

6.4.7.1 Elements of MySQL Enterprise Firewall

MySQL Enterprise Firewall is based on a plugin library that includes these elements:

- A server-side plugin named `MYSQL_FIREWALL` examines SQL statements before they execute and, based on the registered firewall profiles, renders a decision whether to execute or reject each statement.
- The `MYSQL_FIREWALL` plugin, along with server-side plugins named `MYSQL_FIREWALL_USERS` and `MYSQL_FIREWALL_WHITELIST` implement Performance Schema and `INFORMATION_SCHEMA` tables that provide views into the registered profiles.
- Profiles are cached in memory for better performance. Tables in the `mysql` system database provide backing storage of firewall data for persistence of profiles across server restarts.

- Stored procedures perform tasks such as registering firewall profiles, establishing their operational mode, and managing transfer of firewall data between the cache and persistent storage.
- Administrative functions provide an API for lower-level tasks such as synchronizing the cache with persistent storage.
- System variables enable firewall configuration and status variables provide runtime operational information.
- The `FIREWALL_ADMIN` and `FIREWALL_USER` privileges enable users to administer firewall rules for any user, and their own firewall rules, respectively.
- The `FIREWALL_EXEMPT` privilege (available as of MySQL 8.0.27) exempts a user from firewall restrictions. This is useful, for example, for any database administrator who configures the firewall, to avoid the possibility of a misconfiguration causing even the administrator to be locked out and unable to execute statements.

6.4.7.2 Installing or Uninstalling MySQL Enterprise Firewall

MySQL Enterprise Firewall installation is a one-time operation that installs the elements described in Section 6.4.7.1, “Elements of MySQL Enterprise Firewall”. Installation can be performed using a graphical interface or manually:

- On Windows, MySQL Installer includes an option to enable MySQL Enterprise Firewall for you.
- MySQL Workbench 6.3.4 or higher can install MySQL Enterprise Firewall, enable or disable an installed firewall, or uninstall the firewall.
- Manual MySQL Enterprise Firewall installation involves running a script located in the `share` directory of your MySQL installation.

Important

Read this entire section before following its instructions. Parts of the procedure differ depending on your environment.

Note

If installed, MySQL Enterprise Firewall involves some minimal overhead even when disabled. To avoid this overhead, do not install the firewall unless you plan to use it.

For usage instructions, see Section 6.4.7.3, “Using MySQL Enterprise Firewall”. For reference information, see Section 6.4.7.4, “MySQL Enterprise Firewall Reference”.

- Installing MySQL Enterprise Firewall
- Uninstalling MySQL Enterprise Firewall

Installing MySQL Enterprise Firewall

If MySQL Enterprise Firewall is already installed from an older version of MySQL, uninstall it using the instructions given later in this section and then restart your server before installing the current version. In this case, it is also necessary to register your configuration again.

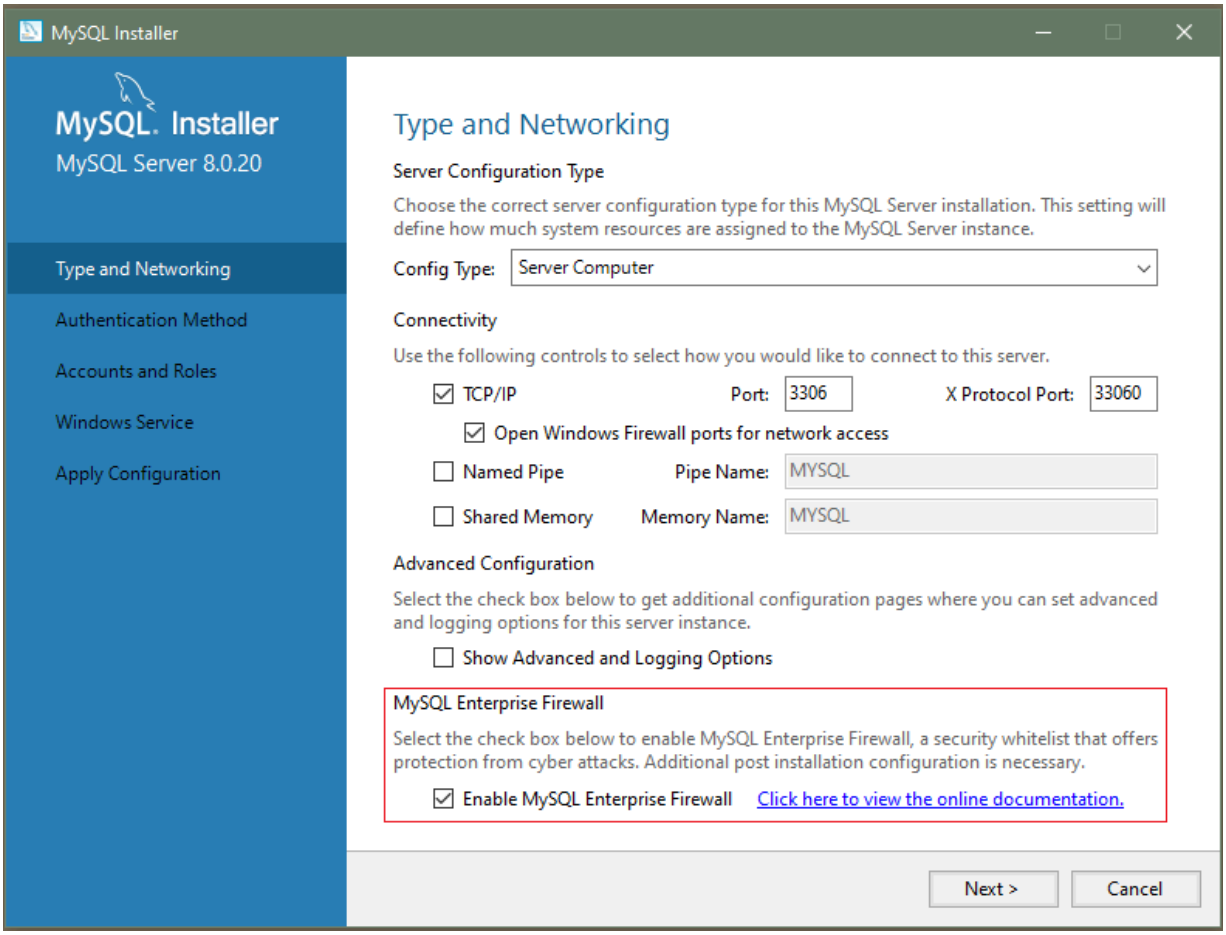
On Windows, you can use MySQL Installer to install MySQL Enterprise Firewall, as shown in Figure 6.2, “MySQL Enterprise Firewall Installation on Windows”. Check the **Enable MySQL Enterprise Firewall** check box. (**Open Firewall port for network access** has a different purpose. It refers to Windows Firewall and controls whether Windows blocks the TCP/IP port on which the MySQL server listens for client connections.)

Important

There is an issue for MySQL 8.0.19 installed using MySQL Installer that prevents the server from starting if MySQL Enterprise Firewall is selected during the server configuration steps. If the server startup operation fails, click **Cancel** to end the configuration process and return to the dashboard. You must uninstall the server.

The workaround is to run MySQL Installer without MySQL Enterprise Firewall selected. (That is, do not select the **Enable MySQL Enterprise Firewall** check box.) Then install MySQL Enterprise Firewall afterward using the instructions for manual installation later in this section. This problem is corrected in MySQL 8.0.20.

Figure 6.2 MySQL Enterprise Firewall Installation on Windows



To install MySQL Enterprise Firewall using MySQL Workbench 6.3.4 or higher, see MySQL Enterprise Firewall Interface.

To install MySQL Enterprise Firewall manually, look in the `share` directory of your MySQL installation and choose the script that is appropriate for your platform. The available scripts differ in the suffix used to refer to the plugin library file:

- `win_install_firewall.sql`: Choose this script for Windows systems that use `.dll` as the file name suffix.
- `linux_install_firewall.sql`: Choose this script for Linux and similar systems that use `.so` as the file name suffix.

The installation script creates stored procedures in the default database, so choose a database to use. Then run the script as follows, naming the chosen database on the command line. The example here uses the `mysql` system database and the Linux installation script. Make the appropriate substitutions for your system.

```
$> mysql -u root -p mysql < linux_install_firewall.sql
Enter password: (enter root password here)
```

Note

To use MySQL Enterprise Firewall in the context of source/replica replication, Group Replication, or InnoDB Cluster, you must prepare the replica nodes prior to running the installation script on the source node. This is necessary because the `INSTALL PLUGIN` statements in the script are not replicated.

1. On each replica node, extract the `INSTALL PLUGIN` statements from the installation script and execute them manually.
2. On the source node, run the installation script as described previously.

Installing MySQL Enterprise Firewall either using a graphical interface or manually should enable the firewall. To verify that, connect to the server and execute this statement:

```
mysql> SHOW GLOBAL VARIABLES LIKE 'mysql_firewall_mode';
+---------------------+-------+
| Variable_name       | Value |
+---------------------+-------+
| mysql_firewall_mode | ON    |
+---------------------+-------+
```

If the plugin fails to initialize, check the server error log for diagnostic messages.

Uninstalling MySQL Enterprise Firewall

MySQL Enterprise Firewall can be uninstalled using MySQL Workbench or manually.

To uninstall MySQL Enterprise Firewall using MySQL Workbench 6.3.4 or higher, see MySQL Enterprise Firewall Interface, in Chapter 31, *MySQL Workbench*.

To uninstall MySQL Enterprise Firewall manually, execute the following statements. Statements use `IF EXISTS` because, depending on the previously installed firewall version, some objects might not exist or might be dropped implicitly by uninstalling the plugin that installed them.

```
DROP TABLE IF EXISTS mysql.firewall_group_allowlist;
DROP TABLE IF EXISTS mysql.firewall_groups;
DROP TABLE IF EXISTS mysql.firewall_membership;
DROP TABLE IF EXISTS mysql.firewall_users;
DROP TABLE IF EXISTS mysql.firewall_whitelist;

UNINSTALL PLUGIN MYSQL_FIREWALL;
UNINSTALL PLUGIN MYSQL_FIREWALL_USERS;
UNINSTALL PLUGIN MYSQL_FIREWALL_WHITELIST;

DROP FUNCTION IF EXISTS firewall_group_delist;
DROP FUNCTION IF EXISTS firewall_group_enlist;
DROP FUNCTION IF EXISTS mysql_firewall_flush_status;
DROP FUNCTION IF EXISTS normalize_statement;
DROP FUNCTION IF EXISTS read_firewall_group_allowlist;
DROP FUNCTION IF EXISTS read_firewall_groups;
DROP FUNCTION IF EXISTS read_firewall_users;
DROP FUNCTION IF EXISTS read_firewall_whitelist;
DROP FUNCTION IF EXISTS set_firewall_group_mode;
DROP FUNCTION IF EXISTS set_firewall_mode;

DROP PROCEDURE IF EXISTS mysql.sp_firewall_group_delist;
DROP PROCEDURE IF EXISTS mysql.sp_firewall_group_enlist;
DROP PROCEDURE IF EXISTS mysql.sp_reload_firewall_group_rules;
```

```
DROP PROCEDURE IF EXISTS mysql.sp_reload_firewall_rules;
DROP PROCEDURE IF EXISTS mysql.sp_set_firewall_group_mode;
DROP PROCEDURE IF EXISTS mysql.sp_set_firewall_group_mode_and_user;
DROP PROCEDURE IF EXISTS mysql.sp_set_firewall_mode;
```

6.4.7.3 Using MySQL Enterprise Firewall

Before using MySQL Enterprise Firewall, install it according to the instructions provided in Section 6.4.7.2, “Installing or Uninstalling MySQL Enterprise Firewall”.

This section describes how to configure MySQL Enterprise Firewall using SQL statements. Alternatively, MySQL Workbench 6.3.4 or higher provides a graphical interface for firewall control. See MySQL Enterprise Firewall Interface.

- Enabling or Disabling the Firewall
- Assigning Firewall Privileges
- Firewall Concepts
- Registering Firewall Group Profiles
- Registering Firewall Account Profiles
- Monitoring the Firewall
- Migrating Account Profiles to Group Profiles

Enabling or Disabling the Firewall

To enable or disable the firewall, set the `mysql_firewall_mode` system variable. By default, this variable is enabled when the firewall is installed. To control the initial firewall state explicitly, you can set the variable at server startup. For example, to enable the firewall in an option file, use these lines:

```
[mysqld]
mysql_firewall_mode=ON
```

After modifying `my.cnf`, restart the server to cause the new setting to take effect.

Alternatively, to set and persist the firewall setting at runtime:

```
SET PERSIST mysql_firewall_mode = OFF;
SET PERSIST mysql_firewall_mode = ON;
```

`SET PERSIST` sets a value for the running MySQL instance. It also saves the value, causing it to carry over to subsequent server restarts. To change a value for the running MySQL instance without having it carry over to subsequent restarts, use the `GLOBAL` keyword rather than `PERSIST`. See Section 13.7.6.1, “SET Syntax for Variable Assignment”.

Assigning Firewall Privileges

With the firewall installed, grant the appropriate privileges to the MySQL account or accounts to be used for administering it. The privileges depend on which firewall operations an account should be permitted to perform:

- Grant the `FIREWALL_EXEMPT` privilege (available as of MySQL 8.0.27) to any account that should be exempt from firewall restrictions. This is useful, for example, for a database administrator who configures the firewall, to avoid the possibility of a misconfiguration causing even the administrator to be locked out and unable to execute statements.
- Grant the `FIREWALL_ADMIN` privilege to any account that should have full administrative firewall access. (Some administrative firewall functions can be invoked by accounts that have `FIREWALL_ADMIN` *or* the deprecated `SUPER` privilege, as indicated in the individual function descriptions.)

- Grant the `FIREWALL_USER` privilege to any account that should have administrative access only for its own firewall rules.
- Grant the `EXECUTE` privilege for the firewall stored procedures in the `mysql` system database. These may invoke administrative functions, so stored procedure access also requires the privileges indicated earlier that are needed for those functions.

Note

The `FIREWALL_EXEMPT`, `FIREWALL_ADMIN`, and `FIREWALL_USER` privileges can be granted only while the firewall is installed because the `MYSQL_FIREWALL` plugin defines those privileges.

Firewall Concepts

The MySQL server permits clients to connect and receives from them SQL statements to be executed. If the firewall is enabled, the server passes to it each incoming statement that does not immediately fail with a syntax error. Based on whether the firewall accepts the statement, the server executes it or returns an error to the client. This section describes how the firewall accomplishes the task of accepting or rejecting statements.

- Firewall Profiles
- Firewall Statement Matching
- Profile Operational Modes
- Firewall Statement Handling When Multiple Profiles Apply

Firewall Profiles

The firewall uses a registry of profiles that determine whether to permit statement execution. Profiles have these attributes:

- An allowlist. The allowlist is the set of rules that defines which statements are acceptable to the profile.
- A current operational mode. The mode enables the profile to be used in different ways. For example: the profile can be placed in training mode to establish the allowlist; the allowlist can be used for restricting statement execution or intrusion detection; the profile can be disabled entirely.
- A scope of applicability. The scope indicates which client connections the profile applies to:
  - The firewall supports account-based profiles such that each profile matches a particular client account (client user name and host name combination). For example, you can register one account profile for which the allowlist applies to connections originating from `admin@localhost` and another account profile for which the allowlist applies to connections originating from `myapp@apphost.example.com`.
  - As of MySQL 8.0.23, the firewall supports group profiles that can have multiple accounts as members, with the profile allowlist applying equally to all members. Group profiles enable easier administration and greater flexibility for deployments that require applying a given set of allowlist rules to multiple accounts.

Initially, no profiles exist, so by default, the firewall accepts all statements and has no effect on which statements MySQL accounts can execute. To apply firewall protective capabilities, explicit action is required:

- Register one or more profiles with the firewall.
- Train the firewall by establishing the allowlist for each profile; that is, the types of statements the profile permits clients to execute.

- Place the trained profiles in protecting mode to harden MySQL against unauthorized statement execution:
  - MySQL associates each client session with a specific user name and host name combination. This combination is the *session account*.
  - For each client connection, the firewall uses the session account to determine which profiles apply to handling incoming statements from the client.

    The firewall accepts only statements permitted by the applicable profile allowlists.

Most firewall principles apply identically to group profiles and account profiles. The two types of profiles differ in these respects:

- An account profile allowlist applies only to a single account. A group profile allowlist applies when the session account matches any account that is a member of the group.
- To apply an allowlist to multiple accounts using account profiles, it is necessary to register one profile per account and duplicate the allowlist across each profile. This entails training each account profile individually because each one must be trained using the single account to which it applies.

  A group profile allowlist applies to multiple accounts, with no need to duplicate it for each account. A group profile can be trained using any or all of the group member accounts, or training can be limited to any single member. Either way, the allowlist applies to all members.
- Account profile names are based on specific user name and host name combinations that depend on which clients connect to the MySQL server. Group profile names are chosen by the firewall administrator with no constraints other than that their length must be from 1 to 288 characters.

Note

Due to the advantages of group profiles over account profiles, and because a group profile with a single member account is logically equivalent to an account profile for that account, it is recommended that all new firewall profiles be created as group profiles. Account profiles are deprecated as of MySQL 8.0.26 and subject to removal in a future MySQL version. For assistance converting existing account profiles, see Migrating Account Profiles to Group Profiles.

The profile-based protection afforded by the firewall enables implementation of strategies such as these:

- If an application has unique protection requirements, configure it to use an account not used for any other purpose and set up a group profile or account profile for that account.
- If related applications share protection requirements, associate each application with its own account, then add these application accounts as members of the same group profile. Alternatively, configure all the applications to use the same account and associate them with an account profile for that account.

Firewall Statement Matching

Statement matching performed by the firewall does not use SQL statements as received from clients. Instead, the server converts incoming statements to normalized digest form and firewall operation uses these digests. The benefit of statement normalization is that it enables similar statements to be grouped and recognized using a single pattern. For example, these statements are distinct from each other:

```
SELECT first_name, last_name FROM customer WHERE customer_id = 1;
select first_name, last_name from customer where customer_id = 99;
SELECT first_name, last_name FROM customer WHERE customer_id = 143;
```

But all of them have the same normalized digest form:

```
SELECT `first_name` , `last_name` FROM `customer` WHERE `customer_id` = ?
```

By using normalization, firewall allowlists can store digests that each match many different statements received from clients. For more information about normalization and digests, see Section 27.10, “Performance Schema Statement Digests and Sampling”.

Warning

Setting the `max_digest_length` system variable to zero disables digest production, which also disables server functionality that requires digests, such as MySQL Enterprise Firewall.

Profile Operational Modes

Each profile registered with the firewall has its own operational mode, chosen from these values:

- `OFF`: This mode disables the profile. The firewall considers it inactive and ignores it.
- `RECORDING`: This is the firewall training mode. Incoming statements received from a client that matches the profile are considered acceptable for the profile and become part of its “fingerprint.” The firewall records the normalized digest form of each statement to learn the acceptable statement patterns for the profile. Each pattern is a rule, and the union of the rules is the profile allowlist.

  A difference between group and account profiles is that statement recording for a group profile can be limited to statements received from a single group member (the training member).
- `PROTECTING`: In this mode, the profile allows or prevents statement execution. The firewall matches incoming statements against the profile allowlist, accepting only statements that match and rejecting those that do not. After training a profile in `RECORDING` mode, switch it to `PROTECTING` mode to harden MySQL against access by statements that deviate from the allowlist. If the `mysql_firewall_trace` system variable is enabled, the firewall also writes rejected statements to the error log.
- `DETECTING`: This mode detects but not does not block intrusions (statements that are suspicious because they match nothing in the profile allowlist). In `DETECTING` mode, the firewall writes suspicious statements to the error log but accepts them without denying access.

When a profile is assigned any of the preceding mode values, the firewall stores the mode in the profile. Firewall mode-setting operations also permit a mode value of `RESET`, but this value is not stored: setting a profile to `RESET` mode causes the firewall to delete all rules for the profile and set its mode to `OFF`.

Note

Messages written to the error log in `DETECTING` mode or because `mysql_firewall_trace` is enabled are written as Notes, which are information messages. To ensure that such messages appear in the error log and are not discarded, make sure that error-logging verbosity is sufficient to include information messages. For example, if you are using priority-based log filtering, as described in Section 5.4.2.5, “Priority-Based Error Log Filtering (log_filter_internal)”, set the `log_error_verbosity` system variable to a value of 3.

Firewall Statement Handling When Multiple Profiles Apply

For simplicity, later sections that describe how to set up profiles take the perspective that the firewall matches incoming statements from a client against only a single profile, either a group profile or account profile. But firewall operation can be more complex:

- A group profile can include multiple accounts as members.
- An account can be a member of multiple group profiles.

- Multiple profiles can match a given client.

The following description covers the general case of how the firewall operates, when potentially multiple profiles apply to incoming statements.

As previously mentioned, MySQL associates each client session with a specific user name and host name combination known as the *session account*. The firewall matches the session account against registered profiles to determine which profiles apply to handling incoming statements from the session:

- The firewall ignores inactive profiles (profiles with a mode of `OFF`).
- The session account matches every active group profile that includes a member having the same user and host. There can be more than one such group profile.
- The session account matches an active account profile having the same user and host, if there is one. There is at most one such account profile.

In other words, the session account can match 0 or more active group profiles, and 0 or 1 active account profiles. This means that 0, 1, or multiple firewall profiles are applicable to a given session, for which the firewall handles each incoming statement as follows:

- If there is no applicable profile, the firewall imposes no restrictions and accepts the statement.
- If there are applicable profiles, their modes determine statement handling:
  - The firewall records the statement in the allowlist of each applicable profile that is in `RECORDING` mode.
  - The firewall writes the statement to the error log for each applicable profile in `DETECTING` mode for which the statement is suspicious (does not match the profile allowlist).
  - The firewall accepts the statement if at least one applicable profile is in `RECORDING` or `DETECTING` mode (those modes accept all statements), or if the statement matches the allowlist of at least one applicable profile in `PROTECTING` mode. Otherwise, the firewall rejects the statement (and writes it to the error log if the `mysql_firewall_trace` system variable is enabled).

With that description in mind, the next sections revert to the simplicity of the situations when a single group profile or a single account profile apply, and cover how to set up each type of profile.

Registering Firewall Group Profiles

MySQL Enterprise Firewall supports registration of group profiles as of MySQL 8.0.23. A group profile can have multiple accounts as its members. To use a firewall group profile to protect MySQL against incoming statements from a given account, follow these steps:

1. Register the group profile and put it in `RECORDING` mode.
2. Add a member account to the group profile.
3. Connect to the MySQL server using the member account and execute statements to be learned. This trains the group profile and establishes the rules that form the profile allowlist.
4. Add to the group profile any other accounts that are to be group members.
5. Switch the group profile to `PROTECTING` mode. When a client connects to the server using any account that is a member of the group profile, the profile allowlist restricts statement execution.
6. Should additional training be necessary, switch the group profile to `RECORDING` mode again, update its allowlist with new statement patterns, then switch it back to `PROTECTING` mode.

Observe these guidelines for firewall-related account references:

- Take note of the context in which account references occur. To name an account for firewall operations, specify it as a single quoted string (`'user_name@host_name'`). This differs from the usual MySQL convention for statements such as `CREATE USER` and `GRANT`, for which you quote the user and host parts of an account name separately (`'user_name'@'host_name'`).

  The requirement for naming accounts as a single quoted string for firewall operations means that you cannot use accounts that have embedded `@` characters in the user name.

- The firewall assesses statements against accounts represented by actual user and host names as authenticated by the server. When registering accounts in profiles, do not use wildcard characters or netmasks:

  - Suppose that an account named `me@%.example.org` exists and a client uses it to connect to the server from the host `abc.example.org`.
  - The account name contains a `%` wildcard character, but the server authenticates the client as having a user name of `me` and host name of `abc.example.com`, and that is what the firewall sees.
  - Consequently, the account name to use for firewall operations is `me@abc.example.org` rather than `me@%.example.org`.

The following procedure shows how to register a group profile with the firewall, train the firewall to know the acceptable statements for that profile (its allowlist), use the profile to protect MySQL against execution of unacceptable statements, and add and remove group members. The example uses a group profile name of `fwgrp`. The example profile is presumed for use by clients of an application that accesses tables in the `sakila` database (available at https://dev.mysql.com/doc/index-other.html).

Use an administrative MySQL account to perform the steps in this procedure, except those steps designated for execution by member accounts of the firewall group profile. For statements executed by member accounts, the default database should be `sakila`. (You can use a different database by adjusting the instructions accordingly.)

1. If necessary, create the accounts that are to be members of the `fwgrp` group profile and grant them appropriate access privileges. Statements for one member are shown here (choose an appropriate password):

   ```
   CREATE USER 'member1'@'localhost' IDENTIFIED BY 'password';
   GRANT ALL ON sakila.* TO 'member1'@'localhost';
   ```

2. Use the `sp_set_firewall_group_mode()` stored procedure to register the group profile with the firewall and place the profile in `RECORDING` (training) mode:

   ```
   CALL mysql.sp_set_firewall_group_mode('fwgrp', 'RECORDING');
   ```

3. Use the `sp_firewall_group_enlist()` stored procedure to add an initial member account for use in training the group profile allowlist:

   ```
   CALL mysql.sp_firewall_group_enlist('fwgrp', 'member1@localhost');
   ```

4. To train the group profile using the initial member account, connect to the server as `member1` from the server host so that the firewall sees a session account of `member1@localhost`. Then execute some statements to be considered legitimate for the profile. For example:

   ```
   SELECT title, release_year FROM film WHERE film_id = 1;
   UPDATE actor SET last_update = NOW() WHERE actor_id = 1;
   SELECT store_id, COUNT(*) FROM inventory GROUP BY store_id;
   ```

   The firewall receives the statements from the `member1@localhost` account. Because that account is a member of the `fwgrp` profile, which is in `RECORDING` mode, the firewall interprets the statements as applicable to `fwgrp` and records the normalized digest form of the statements as rules in the `fwgrp` allowlist. Those rules then apply to all accounts that are members of `fwgrp`.

Note

Until the `fwgrp` group profile receives statements in `RECORDING` mode, its allowlist is empty, which is equivalent to “deny all.” No statement can match an empty allowlist, which has these implications:

- The group profile cannot be switched to `PROTECTING` mode. It would reject every statement, effectively prohibiting the accounts that are group members from executing any statement.
- The group profile can be switched to `DETECTING` mode. In this case, the profile accepts every statement but logs it as suspicious.

5. At this point, the group profile information is cached, including its name, membership, and allowlist. To see this information, query the Performance Schema firewall tables:

```
mysql> SELECT MODE FROM performance_schema.firewall_groups
       WHERE NAME = 'fwgrp';
+-----------+
| MODE      |
+-----------+
| RECORDING |
+-----------+
mysql> SELECT * FROM performance_schema.firewall_membership
       WHERE GROUP_ID = 'fwgrp' ORDER BY MEMBER_ID;
+----------+-------------------+
| GROUP_ID | MEMBER_ID         |
+----------+-------------------+
| fwgrp    | member1@localhost |
+----------+-------------------+
mysql> SELECT RULE FROM performance_schema.firewall_group_allowlist
       WHERE NAME = 'fwgrp';
+-----------------------------------------------------------------------+
| RULE                                                                  |
+-----------------------------------------------------------------------+
| SELECT @@`version_comment` LIMIT ?                                    |
| UPDATE `actor` SET `last_update` = NOW ( ) WHERE `actor_id` = ?       |
| SELECT `title` , `release_year` FROM `film` WHERE `film_id` = ?       |
| SELECT `store_id` , COUNT ( * ) FROM `inventory` GROUP BY `store_id` |
+-----------------------------------------------------------------------+
```

Note

The `@@version_comment` rule comes from a statement sent automatically by the `mysql` client when you connect to the server.

Important

Train the firewall under conditions matching application use. For example, to determine server characteristics and capabilities, a given MySQL connector might send statements to the server at the beginning of each session. If an application normally is used through that connector, train the firewall using the connector, too. That enables those initial statements to become part of the allowlist for the group profile associated with the application.

6. Invoke `sp_set_firewall_group_mode()` again to switch the group profile to `PROTECTING` mode:

```
CALL mysql.sp_set_firewall_group_mode('fwgrp', 'PROTECTING');
```

Important

Switching the group profile out of `RECORDING` mode synchronizes its cached data to the `mysql` system database tables that provide persistent

> underlying storage. If you do not switch the mode for a profile that is being recorded, the cached data is not written to persistent storage and is lost when the server is restarted.

7. Add to the group profile any other accounts that should be members:

```
CALL mysql.sp_firewall_group_enlist('fwgrp', 'member2@localhost');
CALL mysql.sp_firewall_group_enlist('fwgrp', 'member3@localhost');
CALL mysql.sp_firewall_group_enlist('fwgrp', 'member4@localhost');
```

   The profile allowlist trained using the `member1@localhost` account now also applies to the additional accounts.

8. To verify the updated group membership, query the `firewall_membership` table again:

```
mysql> SELECT * FROM performance_schema.firewall_membership
       WHERE GROUP_ID = 'fwgrp' ORDER BY MEMBER_ID;
+----------+-------------------+
| GROUP_ID | MEMBER_ID         |
+----------+-------------------+
| fwgrp    | member1@localhost |
| fwgrp    | member2@localhost |
| fwgrp    | member3@localhost |
| fwgrp    | member4@localhost |
+----------+-------------------+
```

9. Test the group profile against the firewall by using any account in the group to execute some acceptable and unacceptable statements. The firewall matches each statement from the account against the profile allowlist and accepts or rejects it:

   - This statement is not identical to a training statement but produces the same normalized statement as one of them, so the firewall accepts it:

```
mysql> SELECT title, release_year FROM film WHERE film_id = 98;
+-------------------+--------------+
| title             | release_year |
+-------------------+--------------+
| BRIGHT ENCOUNTERS |         2006 |
+-------------------+--------------+
```

   - These statements match nothing in the allowlist, so the firewall rejects each with an error:

```
mysql> SELECT title, release_year FROM film WHERE film_id = 98 OR TRUE;
ERROR 1045 (28000): Statement was blocked by Firewall
mysql> SHOW TABLES LIKE 'customer%';
ERROR 1045 (28000): Statement was blocked by Firewall
mysql> TRUNCATE TABLE mysql.slow_log;
ERROR 1045 (28000): Statement was blocked by Firewall
```

   - If the `mysql_firewall_trace` system variable is enabled, the firewall also writes rejected statements to the error log. For example:

```
[Note] Plugin MYSQL_FIREWALL reported:
'ACCESS DENIED for 'member1@localhost'. Reason: No match in allowlist.
Statement: TRUNCATE TABLE `mysql` . `slow_log`'
```

     These log messages may be helpful in identifying the source of attacks, should that be necessary.

10. Should members need to be removed from the group profile, use the `sp_firewall_group_delist()` stored procedure rather than `sp_firewall_group_enlist()`:

```
CALL mysql.sp_firewall_group_delist('fwgrp', 'member3@localhost');
```

The firewall group profile now is trained for member accounts. When clients connect using any account in the group and attempt to execute statements, the profile protects MySQL against statements not matched by the profile allowlist.

The procedure just shown added only one member to the group profile before training its allowlist. Doing so provides better control over the training period by limiting which accounts can add new acceptable statements to the allowlist. Should additional training be necessary, you can switch the profile back to `RECORDING` mode:

```
CALL mysql.sp_set_firewall_group_mode('fwgrp', 'RECORDING');
```

However, that enables any member of the group to execute statements and add them to the allowlist. To limit the additional training to a single group member, call `sp_set_firewall_group_mode_and_user()`, which is like `sp_set_firewall_group_mode()` but takes one more argument specifying which account is permitted to train the profile in `RECORDING` mode. For example, to enable training only by `member4@localhost`, do this:

```
CALL mysql.sp_set_firewall_group_mode_and_user('fwgrp', 'RECORDING', 'member4@localhost');
```

That enables additional training by the specified account without having to remove the other group members. They can execute statements, but the statements are not added to the allowlist. (Remember, however, that in `RECORDING` mode the other members can execute *any* statement.)

Note

To avoid unexpected behavior when a particular account is specified as the training account for a group profile, always ensure that account is a member of the group.

After the additional training, set the group profile back to `PROTECTING` mode:

```
CALL mysql.sp_set_firewall_group_mode('fwgrp', 'PROTECTING');
```

The training account established by `sp_set_firewall_group_mode_and_user()` is saved in the group profile, so the firewall remembers it in case more training is needed later. Thus, if you call `sp_set_firewall_group_mode()` (which takes no training account argument), the current profile training account, `member4@localhost`, remains unchanged.

To clear the training account if it actually is desired to enable all group members to perform training in `RECORDING` mode, call `sp_set_firewall_group_mode_and_user()` and pass a `NULL` value for the account argument:

```
CALL mysql.sp_set_firewall_group_mode_and_user('fwgrp', 'RECORDING', NULL);
```

It is possible to detect intrusions by logging nonmatching statements as suspicious without denying access. First, put the group profile in `DETECTING` mode:

```
CALL mysql.sp_set_firewall_group_mode('fwgrp', 'DETECTING');
```

Then, using a member account, execute a statement that does not match the group profile allowlist. In `DETECTING` mode, the firewall permits the nonmatching statement to execute:

```
mysql> SHOW TABLES LIKE 'customer%';
+------------------------------+
| Tables_in_sakila (customer%) |
+------------------------------+
| customer                     |
| customer_list                |
+------------------------------+
```

In addition, the firewall writes a message to the error log:

```
[Note] Plugin MYSQL_FIREWALL reported:
'SUSPICIOUS STATEMENT from 'member1@localhost'. Reason: No match in allowlist.
Statement: SHOW TABLES LIKE ?'
```

To disable a group profile, change its mode to `OFF`:

```
CALL mysql.sp_set_firewall_group_mode(group, 'OFF');
```

To forget all training for a profile and disable it, reset it:

```
CALL mysql.sp_set_firewall_group_mode(group, 'RESET');
```

The reset operation causes the firewall to delete all rules for the profile and set its mode to `OFF`.

Registering Firewall Account Profiles

MySQL Enterprise Firewall enables profiles to be registered that correspond to individual accounts. To use a firewall account profile to protect MySQL against incoming statements from a given account, follow these steps:

1. Register the account profile and put it in `RECORDING` mode.
2. Connect to the MySQL server using the account and execute statements to be learned. This trains the account profile and establishes the rules that form the profile allowlist.
3. Switch the account profile to `PROTECTING` mode. When a client connects to the server using the account, the account profile allowlist restricts statement execution.
4. Should additional training be necessary, switch the account profile to `RECORDING` mode again, update its allowlist with new statement patterns, then switch it back to `PROTECTING` mode.

Observe these guidelines for firewall-related account references:

- Take note of the context in which account references occur. To name an account for firewall operations, specify it as a single quoted string (`'user_name@host_name'`). This differs from the usual MySQL convention for statements such as `CREATE USER` and `GRANT`, for which you quote the user and host parts of an account name separately (`'user_name'@'host_name'`).

  The requirement for naming accounts as a single quoted string for firewall operations means that you cannot use accounts that have embedded `@` characters in the user name.

- The firewall assesses statements against accounts represented by actual user and host names as authenticated by the server. When registering accounts in profiles, do not use wildcard characters or netmasks:

  - Suppose that an account named `me@%.example.org` exists and a client uses it to connect to the server from the host `abc.example.org`.
  - The account name contains a `%` wildcard character, but the server authenticates the client as having a user name of `me` and host name of `abc.example.com`, and that is what the firewall sees.
  - Consequently, the account name to use for firewall operations is `me@abc.example.org` rather than `me@%.example.org`.

The following procedure shows how to register an account profile with the firewall, train the firewall to know the acceptable statements for that profile (its allowlist), and use the profile to protect MySQL against execution of unacceptable statements by the account. The example account, `fwuser@localhost`, is presumed for use by an application that accesses tables in the `sakila` database (available at https://dev.mysql.com/doc/index-other.html).

Use an administrative MySQL account to perform the steps in this procedure, except those steps designated for execution by the `fwuser@localhost` account that corresponds to the account profile registered with the firewall. For statements executed using this account, the default database should be `sakila`. (You can use a different database by adjusting the instructions accordingly.)

1. If necessary, create the account to use for executing statements (choose an appropriate password) and grant it privileges for the `sakila` database:

   ```
   CREATE USER 'fwuser'@'localhost' IDENTIFIED BY 'password';
   ```

```
GRANT ALL ON sakila.* TO 'fwuser'@'localhost';
```

2. Use the `sp_set_firewall_mode()` stored procedure to register the account profile with the firewall and place the profile in `RECORDING` (training) mode:

```
CALL mysql.sp_set_firewall_mode('fwuser@localhost', 'RECORDING');
```

3. To train the registered account profile, connect to the server as `fwuser` from the server host so that the firewall sees a session account of `fwuser@localhost`. Then use the account to execute some statements to be considered legitimate for the profile. For example:

```
SELECT first_name, last_name FROM customer WHERE customer_id = 1;
UPDATE rental SET return_date = NOW() WHERE rental_id = 1;
SELECT get_customer_balance(1, NOW());
```

Because the profile is in `RECORDING` mode, the firewall records the normalized digest form of the statements as rules in the profile allowlist.

Note

Until the `fwuser@localhost` account profile receives statements in `RECORDING` mode, its allowlist is empty, which is equivalent to “deny all.” No statement can match an empty allowlist, which has these implications:

- The account profile cannot be switched to `PROTECTING` mode. It would reject every statement, effectively prohibiting the account from executing any statement.
- The account profile can be switched to `DETECTING` mode. In this case, the profile accepts every statement but logs it as suspicious.

4. At this point, the account profile information is cached. To see this information, query the `INFORMATION_SCHEMA` firewall tables:

```
mysql> SELECT MODE FROM INFORMATION_SCHEMA.MYSQL_FIREWALL_USERS
       WHERE USERHOST = 'fwuser@localhost';
+-----------+
| MODE      |
+-----------+
| RECORDING |
+-----------+
mysql> SELECT RULE FROM INFORMATION_SCHEMA.MYSQL_FIREWALL_WHITELIST
       WHERE USERHOST = 'fwuser@localhost';
+--------------------------------------------------------------------------+
| RULE                                                                     |
+--------------------------------------------------------------------------+
| SELECT `first_name` , `last_name` FROM `customer` WHERE `customer_id` = ?  |
| SELECT `get_customer_balance` ( ? , NOW ( ) )                            |
| UPDATE `rental` SET `return_date` = NOW ( ) WHERE `rental_id` = ?        |
| SELECT @@`version_comment` LIMIT ?                                       |
+--------------------------------------------------------------------------+
```

Note

The `@@version_comment` rule comes from a statement sent automatically by the `mysql` client when you connect to the server.

Important

Train the firewall under conditions matching application use. For example, to determine server characteristics and capabilities, a given MySQL connector might send statements to the server at the beginning of each session. If an application normally is used through that connector, train the firewall using the connector, too. That enables those initial statements to become part of the allowlist for the account profile associated with the application.

5. Invoke `sp_set_firewall_mode()` again, this time switching the account profile to `PROTECTING` mode:

```
CALL mysql.sp_set_firewall_mode('fwuser@localhost', 'PROTECTING');
```

Important

Switching the account profile out of `RECORDING` mode synchronizes its cached data to the `mysql` system database tables that provide persistent underlying storage. If you do not switch the mode for a profile that is being recorded, the cached data is not written to persistent storage and is lost when the server is restarted.

6. Test the account profile by using the account to execute some acceptable and unacceptable statements. The firewall matches each statement from the account against the profile allowlist and accepts or rejects it:

   - This statement is not identical to a training statement but produces the same normalized statement as one of them, so the firewall accepts it:

```
mysql> SELECT first_name, last_name FROM customer WHERE customer_id = '48';
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| ANN        | EVANS     |
+------------+-----------+
```

   - These statements match nothing in the allowlist, so the firewall rejects each with an error:

```
mysql> SELECT first_name, last_name FROM customer WHERE customer_id = 1 OR TRUE;
ERROR 1045 (28000): Statement was blocked by Firewall
mysql> SHOW TABLES LIKE 'customer%';
ERROR 1045 (28000): Statement was blocked by Firewall
mysql> TRUNCATE TABLE mysql.slow_log;
ERROR 1045 (28000): Statement was blocked by Firewall
```

   - If the `mysql_firewall_trace` system variable is enabled, the firewall also writes rejected statements to the error log. For example:

```
[Note] Plugin MYSQL_FIREWALL reported:
'ACCESS DENIED for fwuser@localhost. Reason: No match in allowlist.
Statement: TRUNCATE TABLE `mysql` . `slow_log`'
```

     These log messages may be helpful in identifying the source of attacks, should that be necessary.

The firewall account profile now is trained for the `fwuser@localhost` account. When clients connect using that account and attempt to execute statements, the profile protects MySQL against statements not matched by the profile allowlist.

It is possible to detect intrusions by logging nonmatching statements as suspicious without denying access. First, put the account profile in `DETECTING` mode:

```
CALL mysql.sp_set_firewall_mode('fwuser@localhost', 'DETECTING');
```

Then, using the account, execute a statement that does not match the account profile allowlist. In `DETECTING` mode, the firewall permits the nonmatching statement to execute:

```
mysql> SHOW TABLES LIKE 'customer%';
+-----------------------------+
| Tables_in_sakila (customer%) |
+-----------------------------+
| customer                    |
| customer_list               |
+-----------------------------+
```

In addition, the firewall writes a message to the error log:

```
[Note] Plugin MYSQL_FIREWALL reported:
'SUSPICIOUS STATEMENT from 'fwuser@localhost'. Reason: No match in allowlist.
Statement: SHOW TABLES LIKE ?'
```

To disable an account profile, change its mode to `OFF`:

```
CALL mysql.sp_set_firewall_mode(user, 'OFF');
```

To forget all training for a profile and disable it, reset it:

```
CALL mysql.sp_set_firewall_mode(user, 'RESET');
```

The reset operation causes the firewall to delete all rules for the profile and set its mode to `OFF`.

Monitoring the Firewall

To assess firewall activity, examine its status variables. For example, after performing the procedure shown earlier to train and protect the `fwgrp` group profile, the variables look like this:

```
mysql> SHOW GLOBAL STATUS LIKE 'Firewall%';
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Firewall_access_denied     | 3     |
| Firewall_access_granted    | 4     |
| Firewall_access_suspicious | 1     |
| Firewall_cached_entries    | 4     |
+----------------------------+-------+
```

The variables indicate the number of statements rejected, accepted, logged as suspicious, and added to the cache, respectively. The `Firewall_access_granted` count is 4 because of the `@@version_comment` statement sent by the `mysql` client each of the three times you connected using the registered account, plus the `SHOW TABLES` statement that was not blocked in `DETECTING` mode.

Migrating Account Profiles to Group Profiles

Prior to MySQL 8.0.23, MySQL Enterprise Firewall supports only account profiles that each apply to a single account. As of MySQL 8.0.23, the firewall also supports group profiles that each can apply to multiple accounts. A group profile enables easier administration when the same allowlist is to be applied to multiple accounts: instead of creating one account profile per account and duplicating the allowlist across all those profiles, create a single group profile and make the accounts members of it. The group allowlist then applies to all the accounts.

A group profile with a single member account is logically equivalent to an account profile for that account, so it is possible to administer the firewall using group profiles exclusively, rather than a mix of account and group profiles. For new firewall installations, that is accomplished by uniformly creating new profiles as group profiles and avoiding account profiles.

Due to the greater flexibility offered by group profiles, it is recommended that all new firewall profiles be created as group profiles. Account profiles are deprecated as of MySQL 8.0.26 and subject to removal in a future MySQL version. For upgrades from firewall installations that already contain account profiles, MySQL Enterprise Firewall in MySQL 8.0.26 and higher includes a stored procedure named `sp_migrate_firewall_user_to_group()` to help you convert account profiles to group profiles. To use it, perform the following procedure as a user who has the `FIREWALL_ADMIN` privilege:

1. Identify which account profiles exist by querying the Information Schema `MYSQL_FIREWALL_USERS` table. For example:

   ```
   mysql> SELECT USERHOST FROM INFORMATION_SCHEMA.MYSQL_FIREWALL_USERS;
   +------------------------------+
   | USERHOST                     |
   +------------------------------+
   ```

```
| admin@localhost               |
| local_client@localhost        |
| remote_client@abc.example.com |
+-------------------------------+
```

2. For each account profile identified by the previous step, convert it to a group profile:

```
CALL mysql.sp_migrate_firewall_user_to_group('admin@localhost', 'admins');
CALL mysql.sp_migrate_firewall_user_to_group('local_client@localhost', 'local_clients');
CALL mysql.sp_migrate_firewall_user_to_group('remote_client@localhost', 'remote_clients');
```

   In each case, the account profile must exist and must not currently be in `RECORDING` mode, and the group profile must not already exist. The resulting group profile has the named account as its single enlisted member, which is also set as the group training account. The group profile operational mode is taken from the account profile operational mode.

For additional details about `sp_migrate_firewall_user_to_group()`, see Firewall Miscellaneous Stored Procedures.

6.4.7.4 MySQL Enterprise Firewall Reference

The following sections provide a reference to MySQL Enterprise Firewall elements:

- MySQL Enterprise Firewall Tables
- MySQL Enterprise Firewall Stored Procedures
- MySQL Enterprise Firewall Administrative Functions
- MySQL Enterprise Firewall System Variables
- MySQL Enterprise Firewall Status Variables

MySQL Enterprise Firewall Tables

MySQL Enterprise Firewall maintains profile information on a per-group and per-account basis. It uses tables in the `mysql` system database for persistent storage and `INFORMATION_SCHEMA` or Performance Schema tables to provide views into in-memory cached data. When enabled, the firewall bases operational decisions on the cached data.

- Firewall Group Profile Tables
- Firewall Account Profile Tables

Firewall Group Profile Tables

As of MySQL 8.0.23, MySQL Enterprise Firewall maintains group profile information using tables in the `mysql` system database for persistent storage and Performance Schema tables to provide views into in-memory cached data.

Each system and Performance Schema table is accessible only by accounts that have the `SELECT` privilege for it.

The `mysql.firewall_groups` table lists names and operational modes of registered firewall group profiles. The table has the following columns (with the corresponding Performance Schema `firewall_groups` table having similar but not necessarily identical columns):

- `NAME`

  The group profile name.

- `MODE`

  The current operational mode for the profile. Permitted mode values are `OFF`, `DETECTING`, `PROTECTING`, and `RECORDING`. For details about their meanings, see Firewall Concepts.

- `USERHOST`

  The training account for the group profile, to be used when the profile is in `RECORDING` mode. The value is `NULL`, or a non-`NULL` account that has the format *`user_name@host_name`*:

  - If the value is `NULL`, the firewall records allowlist rules for statements received from any account that is a member of the group.
  - If the value is non-`NULL`, the firewall records allowlist rules only for statements received from the named account (which should be a member of the group).

The `mysql.firewall_group_allowlist` table lists allowlist rules of registered firewall group profiles. The table has the following columns (with the corresponding Performance Schema `firewall_group_allowlist` table having similar but not necessarily identical columns):

- `NAME`

  The group profile name.

- `RULE`

  A normalized statement indicating an acceptable statement pattern for the profile. A profile allowlist is the union of its rules.

- `ID`

  An integer column that is a primary key for the table.

The `mysql.firewall_membership` table lists the members (accounts) of registered firewall group profiles. The table has the following columns (with the corresponding Performance Schema `firewall_membership` table having similar but not necessarily identical columns):

- `GROUP_ID`

  The group profile name.

- `MEMBER_ID`

  The name of an account that is a member of the profile.

Firewall Account Profile Tables

MySQL Enterprise Firewall maintains account profile information using tables in the `mysql` system database for persistent storage and `INFORMATION_SCHEMA` tables to provide views into in-memory cached data.

Each `mysql` system database table is accessible only by accounts that have the `SELECT` privilege for it. The `INFORMATION_SCHEMA` tables are accessible by anyone.

As of MySQL 8.0.26, these tables are deprecated and subject to removal in a future MySQL version. See Migrating Account Profiles to Group Profiles.

The `mysql.firewall_users` table lists names and operational modes of registered firewall account profiles. The table has the following columns (with the corresponding `MYSQL_FIREWALL_USERS` table having similar but not necessarily identical columns):

- `USERHOST`

  The account profile name. Each account name has the format *`user_name@host_name`*.

- `MODE`

  The current operational mode for the profile. Permitted mode values are `OFF`, `DETECTING`, `PROTECTING`, `RECORDING`, and `RESET`. For details about their meanings, see Firewall Concepts.

The `mysql.firewall_whitelist` table lists allowlist rules of registered firewall account profiles. The table has the following columns (with the corresponding `MYSQL_FIREWALL_WHITELIST` table having similar but not necessarily identical columns):

- `USERHOST`

  The account profile name. Each account name has the format *`user_name@host_name`*.

- `RULE`

  A normalized statement indicating an acceptable statement pattern for the profile. A profile allowlist is the union of its rules.

- `ID`

  An integer column that is a primary key for the table. This column was added in MySQL 8.0.12.

MySQL Enterprise Firewall Stored Procedures

MySQL Enterprise Firewall stored procedures perform tasks such as registering profiles with the firewall, establishing their operational mode, and managing transfer of firewall data between the cache and persistent storage. These procedures invoke administrative functions that provide an API for lower-level tasks.

Firewall stored procedures are created in the `mysql` system database. To invoke a firewall stored procedure, either do so while `mysql` is the default database, or qualify the procedure name with the database name. For example:

```
CALL mysql.sp_set_firewall_group_mode(group, mode);
```

- Firewall Group Profile Stored Procedures
- Firewall Account Profile Stored Procedures
- Firewall Miscellaneous Stored Procedures

Firewall Group Profile Stored Procedures

These stored procedures perform management operations on firewall group profiles:

- `sp_firewall_group_delist(`*`group`*`, `*`user`*`)`

  This stored procedure removes an account from a firewall group profile.

  If the call succeeds, the change in group membership is made to both the in-memory cache and persistent storage.

  Arguments:

  - *`group`*: The name of the affected group profile.
  - *`user`*: The account to remove, as a string in *`user_name@host_name`* format.

  Example:

  ```
  CALL sp_firewall_group_delist('g', 'fwuser@localhost');
  ```

  This procedure was added in MySQL 8.0.23.

- `sp_firewall_group_enlist(`*`group`*`, `*`user`*`)`

  This stored procedure adds an account to a firewall group profile. It is not necessary to register the account itself with the firewall before adding the account to the group.

If the call succeeds, the change in group membership is made to both the in-memory cache and persistent storage.

Arguments:

- *`group`*: The name of the affected group profile.
- *`user`*: The account to add, as a string in *`user_name@host_name`* format.

Example:

```
CALL sp_firewall_group_enlist('g', 'fwuser@localhost');
```

This procedure was added in MySQL 8.0.23.

- `sp_reload_firewall_group_rules(`*`group`*`)`

This stored procedure provides control over firewall operation for individual group profiles. The procedure uses firewall administrative functions to reload the in-memory rules for a group profile from the rules stored in the `mysql.firewall_group_allowlist` table.

Arguments:

- *`group`*: The name of the affected group profile.

Example:

```
CALL sp_reload_firewall_group_rules('myapp');
```

Warning

This procedure clears the group profile in-memory allowlist rules before reloading them from persistent storage, and sets the profile mode to `OFF`. If the profile mode was not `OFF` prior to the `sp_reload_firewall_group_rules()` call, use `sp_set_firewall_group_mode()` to restore its previous mode after reloading the rules. For example, if the profile was in `PROTECTING` mode, that is no longer true after calling `sp_reload_firewall_group_rules()` and you must set it to `PROTECTING` again explicitly.

This procedure was added in MySQL 8.0.23.

- `sp_set_firewall_group_mode(`*`group, mode`*`)`

This stored procedure establishes the operational mode for a firewall group profile, after registering the profile with the firewall if it was not already registered. The procedure also invokes firewall administrative functions as necessary to transfer firewall data between the cache and persistent

storage. This procedure may be called even if the `mysql_firewall_mode` system variable is `OFF`, although setting the mode for a profile has no operational effect until the firewall is enabled.

If the profile previously existed, any recording limitation for it remains unchanged. To set or clear the limitation, call `sp_set_firewall_group_mode_and_user()` instead.

Arguments:

- *`group`*: The name of the affected group profile.
- *`mode`*: The operational mode for the profile, as a string. Permitted mode values are `OFF`, `DETECTING`, `PROTECTING`, and `RECORDING`. For details about their meanings, see Firewall Concepts.

Example:

```
CALL sp_set_firewall_group_mode('myapp', 'PROTECTING');
```

This procedure was added in MySQL 8.0.23.

- `sp_set_firewall_group_mode_and_user(`*`group`*`, `*`mode`*`, `*`user`*`)`

This stored procedure registers a group with the firewall and establishes its operational mode, similar to `sp_set_firewall_group_mode()`, but also specifies the training account to be used when the group is in `RECORDING` mode.

Arguments:

- *`group`*: The name of the affected group profile.
- *`mode`*: The operational mode for the profile, as a string. Permitted mode values are `OFF`, `DETECTING`, `PROTECTING`, and `RECORDING`. For details about their meanings, see Firewall Concepts.
- *`user`*: The training account for the group profile, to be used when the profile is in `RECORDING` mode. The value is `NULL`, or a non-`NULL` account that has the format *`user_name@host_name`*:
  - If the value is `NULL`, the firewall records allowlist rules for statements received from any account that is a member of the group.
  - If the value is non-`NULL`, the firewall records allowlist rules only for statements received from the named account (which should be a member of the group).

Example:

```
CALL sp_set_firewall_group_mode_and_user('myapp', 'RECORDING', 'myapp_user1@localhost');
```

This procedure was added in MySQL 8.0.23.

Firewall Account Profile Stored Procedures

These stored procedures perform management operations on firewall account profiles:

- `sp_reload_firewall_rules(`*`user`*`)`

This stored procedure provides control over firewall operation for individual account profiles. The procedure uses firewall administrative functions to reload the in-memory rules for an account profile from the rules stored in the `mysql.firewall_whitelist` table.

Arguments:

- *`user`*: The name of the affected account profile, as a string in *`user_name@host_name`* format.

Example:

```
CALL mysql.sp_reload_firewall_rules('fwuser@localhost');
```

Warning

This procedure clears the account profile in-memory allowlist rules before reloading them from persistent storage, and sets the profile mode to `OFF`. If the profile mode was not `OFF` prior to the `sp_reload_firewall_rules()` call, use `sp_set_firewall_mode()` to restore its previous mode after reloading the rules. For example, if the profile was in `PROTECTING` mode, that is no longer true after calling `sp_reload_firewall_rules()` and you must set it to `PROTECTING` again explicitly.

As of MySQL 8.0.26, this procedure is deprecated and subject to removal in a future MySQL version. See Migrating Account Profiles to Group Profiles.

- `sp_set_firewall_mode(`*`user`*`, `*`mode`*`)`

  This stored procedure establishes the operational mode for a firewall account profile, after registering the profile with the firewall if it was not already registered. The procedure also invokes firewall administrative functions as necessary to transfer firewall data between the cache and persistent storage. This procedure may be called even if the `mysql_firewall_mode` system variable is `OFF`, although setting the mode for a profile has no operational effect until the firewall is enabled.

  Arguments:

  - *`user`*: The name of the affected account profile, as a string in *`user_name@host_name`* format.
  - *`mode`*: The operational mode for the profile, as a string. Permitted mode values are `OFF`, `DETECTING`, `PROTECTING`, `RECORDING`, and `RESET`. For details about their meanings, see Firewall Concepts.

  Switching an account profile to any mode but `RECORDING` synchronizes its firewall cache data to the `mysql` system database tables that provide persistent underlying storage. Switching the mode from `OFF` to `RECORDING` reloads the allowlist from the `mysql.firewall_whitelist` table into the cache.

  If an account profile has an empty allowlist, its mode cannot be set to `PROTECTING` because the profile would reject every statement, effectively prohibiting the account from executing statements. In response to such a mode-setting attempt, the firewall produces a diagnostic message that is returned as a result set rather than as an SQL error:

```
mysql> CALL mysql.sp_set_firewall_mode('a@b','PROTECTING');
+-----------------------------------------------------------------+
| set_firewall_mode(arg_userhost, arg_mode)                       |
+-----------------------------------------------------------------+
| ERROR: PROTECTING mode requested for a@b but the allowlist is empty. |
+-----------------------------------------------------------------+
```

  As of MySQL 8.0.26, this procedure is deprecated and subject to removal in a future MySQL version. See Migrating Account Profiles to Group Profiles.

Firewall Miscellaneous Stored Procedures

These stored procedures perform miscellaneous firewall management operations.

- `sp_migrate_firewall_user_to_group(`*`user`*`, `*`group`*`)`

  As of MySQL 8.0.26, account profiles are deprecated because group profiles can do anything account profiles can do. The `sp_migrate_firewall_user_to_group()` stored procedure

converts a firewall account profile to a group profile with the account as its single enlisted member. The conversion procedure is discussed in Migrating Account Profiles to Group Profiles.

This routine requires the `FIREWALL_ADMIN` privilege.

Arguments:

- *`user`*: The name of the account profile to convert to a group profile, as a string in *`user_name@host_name`* format. The account profile must exist, and must not currently be in `RECORDING` mode.
- *`group`*: The name of the new group profile, which must not already exist. The new group profile has the named account as its single enlisted member, and that member is set as the group training account. The group profile operational mode is taken from the account profile operational mode.

Example:

```
CALL sp_migrate_firewall_user_to_group('fwuser@localhost', 'mygroup);
```

This procedure was added in MySQL 8.0.26.

MySQL Enterprise Firewall Administrative Functions

MySQL Enterprise Firewall administrative functions provide an API for lower-level tasks such as synchronizing the firewall cache with the underlying system tables.

Under normal operation, these functions are invoked by the firewall stored procedures, not directly by users. For that reason, these function descriptions do not include details such as information about their arguments and return types.

- Firewall Group Profile Functions
- Firewall Account Profile Functions
- Firewall Miscellaneous Functions

Firewall Group Profile Functions

These functions perform management operations on firewall group profiles:

- `firewall_group_delist(`*`group`*`, `*`user`*`)`

  This function removes an account from a group profile. It requires the `FIREWALL_ADMIN` privilege.

  Example:

  ```
  SELECT firewall_group_delist('g', 'fwuser@localhost');
  ```

  This function was added in MySQL 8.0.23.

- `firewall_group_enlist(`*`group`*`, `*`user`*`)`

  This function adds an account to a group profile. It requires the `FIREWALL_ADMIN` privilege.

  It is not necessary to register the account itself with the firewall before adding the account to the group.

  Example:

  ```
  SELECT firewall_group_enlist('g', 'fwuser@localhost');
  ```

  This function was added in MySQL 8.0.23.

- `read_firewall_group_allowlist(`*`group`*`, `*`rule`*`)`

This aggregate function updates the recorded-statement cache for the named group profile through a `SELECT` statement on the `mysql.firewall_group_allowlist` table. It requires the `FIREWALL_ADMIN` privilege.

Example:

```
SELECT read_firewall_group_allowlist('my_fw_group', fgw.rule)
FROM mysql.firewall_group_allowlist AS fgw
WHERE NAME = 'my_fw_group';
```

This function was added in MySQL 8.0.23.

- `read_firewall_groups(group, mode, user)`

  This aggregate function updates the firewall group profile cache through a `SELECT` statement on the `mysql.firewall_groups` table. It requires the `FIREWALL_ADMIN` privilege.

  Example:

  ```
  SELECT read_firewall_groups('g', 'RECORDING', 'fwuser@localhost')
  FROM mysql.firewall_groups;
  ```

  This function was added in MySQL 8.0.23.

- `set_firewall_group_mode(group, mode[, user])`

  This function manages the group profile cache, establishes the profile operational mode, and optionally specifies the profile training account. It requires the `FIREWALL_ADMIN` privilege.

  If the optional *`user`* argument is not given, any previous *`user`* setting for the profile remains unchanged. To change the setting, call the function with a third argument.

  If the optional *`user`* argument is given, it specifies the training account for the group profile, to be used when the profile is in `RECORDING` mode. The value is `NULL`, or a non-`NULL` account that has the format *`user_name@host_name`*:

  - If the value is `NULL`, the firewall records allowlist rules for statements received from any account that is a member of the group.
  - If the value is non-`NULL`, the firewall records allowlist rules only for statements received from the named account (which should be a member of the group).

  Example:

  ```
  SELECT set_firewall_group_mode('g', 'DETECTING');
  ```

  This function was added in MySQL 8.0.23.

Firewall Account Profile Functions

These functions perform management operations on firewall account profiles:

- `read_firewall_users(user, mode)`

  This aggregate function updates the firewall account profile cache through a `SELECT` statement on the `mysql.firewall_users` table. It requires the `FIREWALL_ADMIN` privilege or the deprecated `SUPER` privilege.

  Example:

  ```
  SELECT read_firewall_users('fwuser@localhost', 'RECORDING')
  FROM mysql.firewall_users;
  ```

As of MySQL 8.0.26, this function is deprecated and subject to removal in a future MySQL version. See Migrating Account Profiles to Group Profiles.

- `read_firewall_whitelist(user, rule)`

  This aggregate function updates the recorded-statement cache for the named account profile through a `SELECT` statement on the `mysql.firewall_whitelist` table. It requires the `FIREWALL_ADMIN` privilege or the deprecated `SUPER` privilege.

  Example:

  ```
  SELECT read_firewall_whitelist('fwuser@localhost', fw.rule)
  FROM mysql.firewall_whitelist AS fw
  WHERE USERHOST = 'fwuser@localhost';
  ```

  As of MySQL 8.0.26, this function is deprecated and subject to removal in a future MySQL version. See Migrating Account Profiles to Group Profiles.

- `set_firewall_mode(user, mode)`

  This function manages the account profile cache and establishes the profile operational mode. It requires the `FIREWALL_ADMIN` privilege or the deprecated `SUPER` privilege.

  Example:

  ```
  SELECT set_firewall_mode('fwuser@localhost', 'RECORDING');
  ```

  As of MySQL 8.0.26, this function is deprecated and subject to removal in a future MySQL version. See Migrating Account Profiles to Group Profiles.

Firewall Miscellaneous Functions

These functions perform miscellaneous firewall operations:

- `mysql_firewall_flush_status()`

  This function resets several firewall status variables to 0:

  - `Firewall_access_denied`
  - `Firewall_access_granted`
  - `Firewall_access_suspicious`

  This function requires the `FIREWALL_ADMIN` privilege or the deprecated `SUPER` privilege.

  Example:

  ```
  SELECT mysql_firewall_flush_status();
  ```

- `normalize_statement(stmt)`

  This function normalizes an SQL statement into the digest form used for allowlist rules. It requires the `FIREWALL_ADMIN` privilege or the deprecated `SUPER` privilege.

  Example:

  ```
  SELECT normalize_statement('SELECT * FROM t1 WHERE c1 > 2');
  ```

Note

The same digest functionality is available outside firewall context using the `STATEMENT_DIGEST_TEXT()` SQL function.

MySQL Enterprise Firewall System Variables

MySQL Enterprise Firewall supports the following system variables. Use them to configure firewall operation. These variables are unavailable unless the firewall is installed (see Section 6.4.7.2, "Installing or Uninstalling MySQL Enterprise Firewall").

- `mysql_firewall_mode`

Command-Line Format	`--mysql-firewall-mode[={OFF|ON}]`
System Variable	`mysql_firewall_mode`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`ON`

Whether MySQL Enterprise Firewall is enabled (the default) or disabled.

- `mysql_firewall_trace`

Command-Line Format	`--mysql-firewall-trace[={OFF|ON}]`
System Variable	`mysql_firewall_trace`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`OFF`

Whether the MySQL Enterprise Firewall trace is enabled or disabled (the default). When `mysql_firewall_trace` is enabled, for `PROTECTING` mode, the firewall writes rejected statements to the error log.

MySQL Enterprise Firewall Status Variables

MySQL Enterprise Firewall supports the following status variables. Use them to obtain information about firewall operational status. These variables are unavailable unless the firewall is installed (see Section 6.4.7.2, "Installing or Uninstalling MySQL Enterprise Firewall"). Firewall status variables are set to 0 whenever the `MYSQL_FIREWALL` plugin is installed or the server is started. Many of them are reset to zero by the `mysql_firewall_flush_status()` function (see MySQL Enterprise Firewall Administrative Functions).

- `Firewall_access_denied`

  The number of statements rejected by MySQL Enterprise Firewall.

- `Firewall_access_granted`

  The number of statements accepted by MySQL Enterprise Firewall.

- `Firewall_access_suspicious`

  The number of statements logged by MySQL Enterprise Firewall as suspicious for users who are in `DETECTING` mode.

- `Firewall_cached_entries`

The number of statements recorded by MySQL Enterprise Firewall, including duplicates.

6.5 MySQL Enterprise Data Masking and De-Identification

Note

MySQL Enterprise Data Masking and De-Identification is an extension included in MySQL Enterprise Edition, a commercial product. To learn more about commercial products, https://www.mysql.com/products/.

As of MySQL 8.0.13, MySQL Enterprise Edition provides data masking and de-identification capabilities:

- Transformation of existing data to mask it and remove identifying characteristics, such as changing all digits of a credit card number but the last four to `'X'` characters.
- Generation of random data, such as email addresses and payment card numbers.

The way that applications use these capabilities depends on the purpose for which the data is used and who accesses it:

- Applications that use sensitive data may protect it by performing data masking and permitting use of partially masked data for client identification. Example: A call center may ask for clients to provide their last four Social Security number digits.
- Applications that require properly formatted data, but not necessarily the original data, can synthesize sample data. Example: An application developer who is testing data validators but has no access to original data may synthesize random data with the same format.

Example 1:

Medical research facilities can hold patient data that comprises a mix of personal and medical data. This may include genetic sequences (long strings), test results stored in JSON format, and other data types. Although the data may be used mostly by automated analysis software, access to genome data or test results of particular patients is still possible. In such cases, data masking should be used to render this information not personally identifiable.

Example 2:

A credit card processor company provides a set of services using sensitive data, such as:

- Processing a large number of financial transactions per second.
- Storing a large amount of transaction-related data.
- Protecting transaction-related data with strict requirements for personal data.
- Handling client complaints about transactions using reversible or partially masked data.

A typical transaction may include many types of sensitive information, including:

- Credit card number.
- Transaction type and amount.
- Merchant type.
- Transaction cryptogram (to confirm transaction legitimacy).
- Geolocation of GPS-equipped terminal (for fraud detection).

Those types of information may then be joined within a bank or other card-issuing financial institution with client personal data, such as:

- Full client name (either person or company).
- Address.
- Date of birth.
- Social Security number.
- Email address.
- Phone number.

Various employee roles within both the card processing company and the financial institution require access to that data. Some of these roles may require access only to masked data. Other roles may require access to the original data on a case-to-case basis, which is recorded in audit logs.

Masking and de-identification are core to regulatory compliance, so MySQL Enterprise Data Masking and De-Identification can help application developers satisfy privacy requirements:

- PCI – DSS: Payment Card Data.
- HIPAA: Privacy of Health Data, Health Information Technology for Economic and Clinical Health Act (HITECH Act).
- EU General Data Protection Directive (GDPR): Protection of Personal Data.
- Data Protection Act (UK): Protection of Personal Data.
- Sarbanes Oxley, GLBA, The USA Patriot Act, Identity Theft and Assumption Deterrence Act of 1998.
- FERPA – Student Data, NASD, CA SB1386 and AB 1950, State Data Protection Laws, Basel II.

The following sections describe the elements of MySQL Enterprise Data Masking and De-Identification, discuss how to install and use it, and provide reference information for its elements.

6.5.1 MySQL Enterprise Data Masking and De-Identification Elements

MySQL Enterprise Data Masking and De-Identification is based on a plugin library that implements these elements:

- A server-side plugin named `data_masking`.
- A set of loadable functions provides an SQL-level API for performing masking and de-identification operations. Some of these functions require the `SUPER` privilege.

6.5.2 Installing or Uninstalling MySQL Enterprise Data Masking and De-Identification

This section describes how to install or uninstall MySQL Enterprise Data Masking and De-Identification, which is implemented as a plugin library file containing a plugin and several loadable functions. For general information about installing or uninstalling plugins and loadable functions, see Section 5.6.1, “Installing and Uninstalling Plugins”, and Section 5.7.1, “Installing and Uninstalling Loadable Functions”.

To be usable by the server, the plugin library file must be located in the MySQL plugin directory (the directory named by the `plugin_dir` system variable). If necessary, configure the plugin directory location by setting the value of `plugin_dir` at server startup.

The plugin library file base name is `data_masking`. The file name suffix differs per platform (for example, `.so` for Unix and Unix-like systems, `.dll` for Windows).

To install the MySQL Enterprise Data Masking and De-Identification plugin and functions, use the `INSTALL PLUGIN` and `CREATE FUNCTION` statements, adjusting the `.so` suffix for your platform as necessary:

```
INSTALL PLUGIN data_masking SONAME 'data_masking.so';
CREATE FUNCTION gen_blocklist RETURNS STRING
  SONAME 'data_masking.so';
CREATE FUNCTION gen_dictionary RETURNS STRING
  SONAME 'data_masking.so';
CREATE FUNCTION gen_dictionary_drop RETURNS STRING
  SONAME 'data_masking.so';
CREATE FUNCTION gen_dictionary_load RETURNS STRING
  SONAME 'data_masking.so';
CREATE FUNCTION gen_range RETURNS INTEGER
  SONAME 'data_masking.so';
CREATE FUNCTION gen_rnd_email RETURNS STRING
  SONAME 'data_masking.so';
CREATE FUNCTION gen_rnd_pan RETURNS STRING
  SONAME 'data_masking.so';
CREATE FUNCTION gen_rnd_ssn RETURNS STRING
  SONAME 'data_masking.so';
CREATE FUNCTION gen_rnd_us_phone RETURNS STRING
  SONAME 'data_masking.so';
CREATE FUNCTION mask_inner RETURNS STRING
  SONAME 'data_masking.so';
CREATE FUNCTION mask_outer RETURNS STRING
  SONAME 'data_masking.so';
CREATE FUNCTION mask_pan RETURNS STRING
  SONAME 'data_masking.so';
CREATE FUNCTION mask_pan_relaxed RETURNS STRING
  SONAME 'data_masking.so';
CREATE FUNCTION mask_ssn RETURNS STRING
  SONAME 'data_masking.so';
```

If the plugin and functions are used on a replication source server, install them on all replica servers as well to avoid replication issues.

Once installed as just described, the plugin and functions remain installed until uninstalled. To remove them, use the `UNINSTALL PLUGIN` and `DROP FUNCTION` statements:

```
UNINSTALL PLUGIN data_masking;
DROP FUNCTION gen_blocklist;
DROP FUNCTION gen_dictionary;
DROP FUNCTION gen_dictionary_drop;
DROP FUNCTION gen_dictionary_load;
DROP FUNCTION gen_range;
DROP FUNCTION gen_rnd_email;
DROP FUNCTION gen_rnd_pan;
DROP FUNCTION gen_rnd_ssn;
DROP FUNCTION gen_rnd_us_phone;
DROP FUNCTION mask_inner;
DROP FUNCTION mask_outer;
DROP FUNCTION mask_pan;
DROP FUNCTION mask_pan_relaxed;
DROP FUNCTION mask_ssn;
```

6.5.3 Using MySQL Enterprise Data Masking and De-Identification

Before using MySQL Enterprise Data Masking and De-Identification, install it according to the instructions provided at Section 6.5.2, “Installing or Uninstalling MySQL Enterprise Data Masking and De-Identification”.

To use MySQL Enterprise Data Masking and De-Identification in applications, invoke the functions that are appropriate for the operations you wish to perform. For detailed function descriptions, see Section 6.5.5, “MySQL Enterprise Data Masking and De-Identification Function Descriptions”. This section demonstrates how to use the functions to carry out some representative tasks. It first presents an overview of the available functions, followed by some examples of how the functions might be used in real-world context:

Masking Data to Remove Identifying Characteristics

MySQL provides general-purpose masking functions that mask arbitrary strings, and special-purpose masking functions that mask specific types of values.

General-Purpose Masking Functions

`mask_inner()` and `mask_outer()` are general-purpose functions that mask parts of arbitrary strings based on position within the string:

- `mask_inner()` masks the interior of its string argument, leaving the ends unmasked. Other arguments specify the sizes of the unmasked ends.

```
mysql> SELECT mask_inner('This is a string', 5, 1);
+--------------------------------------+
| mask_inner('This is a string', 5, 1) |
+--------------------------------------+
| This XXXXXXXXXg                      |
+--------------------------------------+
mysql> SELECT mask_inner('This is a string', 1, 5);
+--------------------------------------+
| mask_inner('This is a string', 1, 5) |
+--------------------------------------+
| TXXXXXXXXXtring                      |
+--------------------------------------+
```

- `mask_outer()` does the reverse, masking the ends of its string argument, leaving the interior unmasked. Other arguments specify the sizes of the masked ends.

```
mysql> SELECT mask_outer('This is a string', 5, 1);
+--------------------------------------+
| mask_outer('This is a string', 5, 1) |
+--------------------------------------+
| XXXXXis a strinX                     |
+--------------------------------------+
mysql> SELECT mask_outer('This is a string', 1, 5);
+--------------------------------------+
| mask_outer('This is a string', 1, 5) |
+--------------------------------------+
| Xhis is a sXXXXX                     |
+--------------------------------------+
```

By default, `mask_inner()` and `mask_outer()` use `'X'` as the masking character, but permit an optional masking-character argument:

```
mysql> SELECT mask_inner('This is a string', 5, 1, '*');
+-----------------------------------------+
| mask_inner('This is a string', 5, 1, '*') |
+-----------------------------------------+
| This *********g                         |
+-----------------------------------------+
mysql> SELECT mask_outer('This is a string', 5, 1, '#');
+-----------------------------------------+
| mask_outer('This is a string', 5, 1, '#') |
+-----------------------------------------+
| #####is a strin#                        |
+-----------------------------------------+
```

Special-Purpose Masking Functions

Other masking functions expect a string argument representing a specific type of value and mask it to remove identifying characteristics.

Note

The examples here supply function arguments using the random value generation functions that return the appropriate type of value. For more information about generation functions, see Generating Random Data with Specific Characteristics.

Payment card Primary Account Number masking. Masking functions provide strict and relaxed masking of Primary Account Numbers.

- `mask_pan()` masks all but the last four digits of the number:

```
mysql> SELECT mask_pan(gen_rnd_pan());
+-------------------------+
| mask_pan(gen_rnd_pan()) |
+-------------------------+
| XXXXXXXXXXX2461         |
+-------------------------+
```

- `mask_pan_relaxed()` is similar but does not mask the first six digits that indicate the payment card issuer unmasked:

```
mysql> SELECT mask_pan_relaxed(gen_rnd_pan());
+---------------------------------+
| mask_pan_relaxed(gen_rnd_pan()) |
+---------------------------------+
| 770630XXXXXX0807                |
+---------------------------------+
```

US Social Security number masking. `mask_ssn()` masks all but the last four digits of the number:

```
mysql> SELECT mask_ssn(gen_rnd_ssn());
+-------------------------+
| mask_ssn(gen_rnd_ssn()) |
+-------------------------+
| XXX-XX-1723             |
+-------------------------+
```

Generating Random Data with Specific Characteristics

Several functions generate random values. These values can be used for testing, simulation, and so forth.

`gen_range()` returns a random integer selected from a given range:

```
mysql> SELECT gen_range(1, 10);
+------------------+
| gen_range(1, 10) |
+------------------+
|                6 |
+------------------+
```

`gen_rnd_email()` returns a random email address in the `example.com` domain:

```
mysql> SELECT gen_rnd_email();
+--------------------------+
| gen_rnd_email()          |
+--------------------------+
| ayxnq.xmkpvvy@example.com |
+--------------------------+
```

`gen_rnd_pan()` returns a random payment card Primary Account Number:

```
mysql> SELECT gen_rnd_pan();
```

(The `gen_rnd_pan()` function result is not shown because its return values should be used only for testing purposes, and not for publication. It cannot be guaranteed the number is not assigned to a legitimate payment account.)

`gen_rnd_ssn()` returns a random US Social Security number with the first and second parts each chosen from a range not used for legitimate numbers:

```
mysql> SELECT gen_rnd_ssn();
+---------------+
| gen_rnd_ssn() |
+---------------+
| 912-45-1615   |
+---------------+
```

`gen_rnd_us_phone()` returns a random US phone number in the 555 area code not used for legitimate numbers:

```
mysql> SELECT gen_rnd_us_phone();
+--------------------+
| gen_rnd_us_phone() |
+--------------------+
| 1-555-747-5627     |
+--------------------+
```

Generating Random Data Using Dictionaries

MySQL Enterprise Data Masking and De-Identification enables dictionaries to be used as sources of random values. To use a dictionary, it must first be loaded from a file and given a name. Each loaded dictionary becomes part of the dictionary registry. Items then can be selected from registered dictionaries and used as random values or as replacements for other values.

A valid dictionary file has these characteristics:

- The file contents are plain text, one term per line.
- Empty lines are ignored.
- The file must contain at least one term.

Suppose that a file named `de_cities.txt` contains these city names in Germany:

```
Berlin
Munich
Bremen
```

Also suppose that a file named `us_cities.txt` contains these city names in the United States:

```
Chicago
Houston
Phoenix
El Paso
Detroit
```

Assume that the `secure_file_priv` system variable is set to `/usr/local/mysql/mysql-files`. In that case, copy the dictionary files to that directory so that the MySQL server can access them. Then use `gen_dictionary_load()` to load the dictionaries into the dictionary registry and assign them names:

```
mysql> SELECT gen_dictionary_load('/usr/local/mysql/mysql-files/de_cities.txt', 'DE_Cities');
+-----------------------------------------------------------------------------+
| gen_dictionary_load('/usr/local/mysql/mysql-files/de_cities.txt', 'DE_Cities') |
+-----------------------------------------------------------------------------+
| Dictionary load success                                                     |
+-----------------------------------------------------------------------------+
mysql> SELECT gen_dictionary_load('/usr/local/mysql/mysql-files/us_cities.txt', 'US_Cities');
+-----------------------------------------------------------------------------+
```

```
| gen_dictionary_load('/usr/local/mysql/mysql-files/us_cities.txt', 'US_Cities') |
+--------------------------------------------------------------------------------+
| Dictionary load success                                                        |
+--------------------------------------------------------------------------------+
```

To select a random term from a dictionary, use `gen_dictionary()`:

```
mysql> SELECT gen_dictionary('DE_Cities');
+----------------------------+
| gen_dictionary('DE_Cities') |
+----------------------------+
| Berlin                     |
+----------------------------+
mysql> SELECT gen_dictionary('US_Cities');
+----------------------------+
| gen_dictionary('US_Cities') |
+----------------------------+
| Phoenix                    |
+----------------------------+
```

To select a random term from multiple dictionaries, randomly select one of the dictionaries, then select a term from it:

```
mysql> SELECT gen_dictionary(ELT(gen_range(1,2), 'DE_Cities', 'US_Cities'));
+-------------------------------------------------------------+
| gen_dictionary(ELT(gen_range(1,2), 'DE_Cities', 'US_Cities')) |
+-------------------------------------------------------------+
| Detroit                                                     |
+-------------------------------------------------------------+
mysql> SELECT gen_dictionary(ELT(gen_range(1,2), 'DE_Cities', 'US_Cities'));
+-------------------------------------------------------------+
| gen_dictionary(ELT(gen_range(1,2), 'DE_Cities', 'US_Cities')) |
+-------------------------------------------------------------+
| Bremen                                                      |
+-------------------------------------------------------------+
```

The `gen_blocklist()` function enables a term from one dictionary to be replaced by a term from another dictionary, which effects masking by substitution. Its arguments are the term to replace, the dictionary in which the term appears, and the dictionary from which to choose a replacement. For example, to substitute a US city for a German city, or vice versa, use `gen_blocklist()` like this:

```
mysql> SELECT gen_blocklist('Munich', 'DE_Cities', 'US_Cities');
+-----------------------------------------------+
| gen_blocklist('Munich', 'DE_Cities', 'US_Cities') |
+-----------------------------------------------+
| Houston                                       |
+-----------------------------------------------+
mysql> SELECT gen_blocklist('El Paso', 'US_Cities', 'DE_Cities');
+-----------------------------------------------+
| gen_blocklist('El Paso', 'US_Cities', 'DE_Cities') |
+-----------------------------------------------+
| Bremen                                        |
+-----------------------------------------------+
```

If the term to replace is not in the first dictionary, `gen_blocklist()` returns it unchanged:

```
mysql> SELECT gen_blocklist('Moscow', 'DE_Cities', 'US_Cities');
+-----------------------------------------------+
| gen_blocklist('Moscow', 'DE_Cities', 'US_Cities') |
+-----------------------------------------------+
| Moscow                                        |
+-----------------------------------------------+
```

Using Masked Data for Customer Identification

At customer-service call centers, one common identity verification technique is to ask customers to provide their last four Social Security number (SSN) digits. For example, a customer might say her name is Joanna Bond and that her last four SSN digits are `0007`.

Suppose that a `customer` table containing customer records has these columns:

- `id`: Customer ID number.
- `first_name`: Customer first name.
- `last_name`: Customer last name.
- `ssn`: Customer Social Security number.

For example, the table might be defined as follows:

```
CREATE TABLE customer
(
  id         BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  first_name VARCHAR(40),
  last_name  VARCHAR(40),
  ssn        VARCHAR(11)
);
```

The application used by customer-service representatives to check the customer SSN might execute a query like this:

```
mysql> SELECT id, ssn
mysql> FROM customer
mysql> WHERE first_name = 'Joanna' AND last_name = 'Bond';
+-----+-------------+
| id  | ssn         |
+-----+-------------+
| 786 | 906-39-0007 |
+-----+-------------+
```

However, that exposes the SSN to the customer-service representative, who has no need to see anything but the last four digits. Instead, the application can use this query to display only the masked SSN:

```
mysql> SELECT id, mask_ssn(CONVERT(ssn USING binary)) AS masked_ssn
mysql> FROM customer
mysql> WHERE first_name = 'Joanna' AND last_name = 'Bond';
+-----+-------------+
| id  | masked_ssn  |
+-----+-------------+
| 786 | XXX-XX-0007 |
+-----+-------------+
```

Now the representative sees only what is necessary, and customer privacy is preserved.

Why was the `CONVERT()` function used for the argument to `mask_ssn()`? Because `mask_ssn()` requires an argument of length 11. Thus, even though `ssn` is defined as `VARCHAR(11)`, if the `ssn` column has a multibyte character set, it may appear to be longer than 11 bytes when passed to a loadable function, and an error occurs. Converting the value to a binary string ensures that the function sees an argument of length 11.

A similar technique may be needed for other data masking functions when string arguments do not have a single-byte character set.

Creating Views that Display Masked Data

If masked data from a table is used for multiple queries, it may be convenient to define a view that produces masked data. That way, applications can select from the view without performing masking in individual queries.

For example, a masking view on the `customer` table from the previous section can be defined like this:

```
CREATE VIEW masked_customer AS
SELECT id, first_name, last_name,
mask_ssn(CONVERT(ssn USING binary)) AS masked_ssn
FROM customer;
```

Then the query to look up a customer becomes simpler but still returns masked data:

```
mysql> SELECT id, masked_ssn
mysql> FROM masked_customer
mysql> WHERE first_name = 'Joanna' AND last_name = 'Bond';
+-----+-------------+
| id  | masked_ssn  |
+-----+-------------+
| 786 | XXX-XX-0007 |
+-----+-------------+
```

6.5.4 MySQL Enterprise Data Masking and De-Identification Function Reference

Table 6.45 MySQL Enterprise Data Masking and De-Identification Functions

Name	Description	Introduced	Deprecated
`gen_blacklist()`	Perform dictionary term replacement		8.0.23
`gen_blocklist()`	Perform dictionary term replacement	8.0.23	
`gen_dictionary()`	Return random term from dictionary		
`gen_dictionary_drop()`	Remove dictionary from registry		
`gen_dictionary_load()`	Load dictionary into registry		
`gen_range()`	Generate random number within range		
`gen_rnd_email()`	Generate random email address		
`gen_rnd_pan()`	Generate random payment card Primary Account Number		
`gen_rnd_ssn()`	Generate random US Social Security number		
`gen_rnd_us_phone()`	Generate random US phone number		
`mask_inner()`	Mask interior part of string		
`mask_outer()`	Mask left and right parts of string		
`mask_pan()`	Mask payment card Primary Account Number part of string		
`mask_pan_relaxed()`	Mask payment card Primary Account Number part of string		
`mask_ssn()`	Mask US Social Security number		

6.5.5 MySQL Enterprise Data Masking and De-Identification Function Descriptions

The MySQL Enterprise Data Masking and De-Identification plugin library includes several functions, which may be grouped into these categories:

- Data Masking Functions
- Random Data Generation Functions
- Random Data Dictionary-Based Functions

As of MySQL 8.0.19, these functions support the single-byte `latin1` character set for string arguments and return values. Prior to MySQL 8.0.19, the functions treat string arguments as binary strings (which means they do not distinguish lettercase), and string return values are binary strings. You can see the difference in return value character set as follows:

MySQL 8.0.19 and higher:

```
mysql> SELECT CHARSET(gen_rnd_email());
+--------------------------+
| CHARSET(gen_rnd_email()) |
+--------------------------+
| latin1                   |
+--------------------------+
```

Prior to MySQL 8.0.19:

```
mysql> SELECT CHARSET(gen_rnd_email());
+--------------------------+
| CHARSET(gen_rnd_email()) |
+--------------------------+
| binary                   |
+--------------------------+
```

For any version, if a string return value should be in a different character set, convert it. The following example shows how to convert the result of `gen_rnd_email()` to the `utf8mb4` character set:

```
SET @email = CONVERT(gen_rnd_email() USING utf8mb4);
```

To explicitly produce a binary string (for example, to produce a result like that for MySQL versions prior to 8.0.19), do this:

```
SET @email = CONVERT(gen_rnd_email() USING binary);
```

It may also be necessary to convert string arguments, as illustrated in Using Masked Data for Customer Identification.

If a MySQL Enterprise Data Masking and De-Identification function is invoked from within the `mysql` client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

Data Masking Functions

Each function in this section performs a masking operation on its string argument and returns the masked result.

- `mask_inner(`*`str`*`, `*`margin1`*`, `*`margin2`*` [, `*`mask_char`*`])`

  Masks the interior part of a string, leaving the ends untouched, and returns the result. An optional masking character can be specified.

  Arguments:

  - *`str`*: The string to mask.
  - *`margin1`*: A nonnegative integer that specifies the number of characters on the left end of the string to remain unmasked. If the value is 0, no left end characters remain unmasked.

- *margin2*: A nonnegative integer that specifies the number of characters on the right end of the string to remain unmasked. If the value is 0, no right end characters remain unmasked.
- *mask_char*: (Optional) The single character to use for masking. The default is 'X' if *mask_char* is not given.

  The masking character must be a single-byte character. Attempts to use a multibyte character produce an error.

Return value:

The masked string, or NULL if either margin is negative.

If the sum of the margin values is larger than the argument length, no masking occurs and the argument is returned unchanged.

Example:

```
mysql> SELECT mask_inner('abcdef', 1, 2), mask_inner('abcdef',0, 5);
+---------------------------+--------------------------+
| mask_inner('abcdef', 1, 2) | mask_inner('abcdef',0, 5) |
+---------------------------+--------------------------+
| aXXXef                    | Xbcdef                   |
+---------------------------+--------------------------+
mysql> SELECT mask_inner('abcdef', 1, 2, '*'), mask_inner('abcdef',0, 5, '#');
+---------------------------------+-------------------------------+
| mask_inner('abcdef', 1, 2, '*') | mask_inner('abcdef',0, 5, '#') |
+---------------------------------+-------------------------------+
| a***ef                          | #bcdef                        |
+---------------------------------+-------------------------------+
```

- mask_outer(*str*, *margin1*, *margin2* [, *mask_char*])

Masks the left and right ends of a string, leaving the interior unmasked, and returns the result. An optional masking character can be specified.

Arguments:

- *str*: The string to mask.
- *margin1*: A nonnegative integer that specifies the number of characters on the left end of the string to mask. If the value is 0, no left end characters are masked.
- *margin2*: A nonnegative integer that specifies the number of characters on the right end of the string to mask. If the value is 0, no right end characters are masked.
- *mask_char*: (Optional) The single character to use for masking. The default is 'X' if *mask_char* is not given.

  The masking character must be a single-byte character. Attempts to use a multibyte character produce an error.

Return value:

The masked string, or NULL if either margin is negative.

If the sum of the margin values is larger than the argument length, the entire argument is masked.

Example:

```
mysql> SELECT mask_outer('abcdef', 1, 2), mask_outer('abcdef',0, 5);
+---------------------------+--------------------------+
| mask_outer('abcdef', 1, 2) | mask_outer('abcdef',0, 5) |
+---------------------------+--------------------------+
```

```
| XbcdXX                     | aXXXXX                     |
+----------------------------+----------------------------+
mysql> SELECT mask_outer('abcdef', 1, 2, '*'), mask_outer('abcdef',0, 5, '#');
+--------------------------------+--------------------------------+
| mask_outer('abcdef', 1, 2, '*') | mask_outer('abcdef',0, 5, '#') |
+--------------------------------+--------------------------------+
| *bcd**                         | a#####                         |
+--------------------------------+--------------------------------+
```

- `mask_pan(str)`

  Masks a payment card Primary Account Number and returns the number with all but the last four digits replaced by `'X'` characters.

  Arguments:

  - *`str`*: The string to mask. The string must be a suitable length for the Primary Account Number, but is not otherwise checked.

  Return value:

  The masked payment number as a string. If the argument is shorter than required, it is returned unchanged.

  Example:

```
mysql> SELECT mask_pan(gen_rnd_pan());
+-------------------------+
| mask_pan(gen_rnd_pan()) |
+-------------------------+
| XXXXXXXXXXX9102         |
+-------------------------+
mysql> SELECT mask_pan(gen_rnd_pan(19));
+---------------------------+
| mask_pan(gen_rnd_pan(19)) |
+---------------------------+
| XXXXXXXXXXXXXX8268        |
+---------------------------+
mysql> SELECT mask_pan('a*Z');
+-----------------+
| mask_pan('a*Z') |
+-----------------+
| a*Z             |
+-----------------+
```

- `mask_pan_relaxed(str)`

  Masks a payment card Primary Account Number and returns the number with all but the first six and last four digits replaced by `'X'` characters. The first six digits indicate the payment card issuer.

  Arguments:

  - *`str`*: The string to mask. The string must be a suitable length for the Primary Account Number, but is not otherwise checked.

  Return value:

  The masked payment number as a string. If the argument is shorter than required, it is returned unchanged.

  Example:

```
mysql> SELECT mask_pan_relaxed(gen_rnd_pan());
+---------------------------------+
| mask_pan_relaxed(gen_rnd_pan()) |
+---------------------------------+
| 551279XXXXXX3108                |
```

```
+---------------------------------+
mysql> SELECT mask_pan_relaxed(gen_rnd_pan(19));
+-----------------------------------+
| mask_pan_relaxed(gen_rnd_pan(19)) |
+-----------------------------------+
| 462634XXXXXXXXX6739               |
+-----------------------------------+
mysql> SELECT mask_pan_relaxed('a*Z');
+-------------------------+
| mask_pan_relaxed('a*Z') |
+-------------------------+
| a*Z                     |
+-------------------------+
```

- `mask_ssn(str)`

  Masks a US Social Security number and returns the number with all but the last four digits replaced by `'X'` characters.

  Arguments:

  - *`str`*: The string to mask. The string must be 11 characters long.

  Return value:

  The masked Social Security number as a string, or an error if the argument is not the correct length.

  Example:

```
mysql> SELECT mask_ssn('909-63-6922'), mask_ssn('abcdefghijk');
+-------------------------+-------------------------+
| mask_ssn('909-63-6922') | mask_ssn('abcdefghijk') |
+-------------------------+-------------------------+
| XXX-XX-6922             | XXX-XX-hijk             |
+-------------------------+-------------------------+
mysql> SELECT mask_ssn('909');
ERROR 1123 (HY000): Can't initialize function 'mask_ssn'; MASK_SSN: Error:
String argument width too small
mysql> SELECT mask_ssn('123456789123456789');
ERROR 1123 (HY000): Can't initialize function 'mask_ssn'; MASK_SSN: Error:
String argument width too large
```

Random Data Generation Functions

The functions in this section generate random values for different types of data. When possible, generated values have characteristics reserved for demonstration or test values, to avoid having them mistaken for legitimate data. For example, `gen_rnd_us_phone()` returns a US phone number that uses the 555 area code, which is not assigned to phone numbers in actual use. Individual function descriptions describe any exceptions to this principle.

- `gen_range(lower, upper)`

  Generates a random number chosen from a specified range.

  Arguments:

  - *`lower`*: An integer that specifies the lower boundary of the range.
  - *`upper`*: An integer that specifies the upper boundary of the range, which must not be less than the lower boundary.

  Return value:

  A random integer in the range from *`lower`* to *`upper`*, inclusive, or `NULL` if the *`upper`* argument is less than *`lower`*.

Example:

```
mysql> SELECT gen_range(100, 200), gen_range(-1000, -800);
+--------------------+----------------------+
| gen_range(100, 200) | gen_range(-1000, -800) |
+--------------------+----------------------+
|                 177 |                 -917 |
+--------------------+----------------------+
mysql> SELECT gen_range(1, 0);
+-----------------+
| gen_range(1, 0) |
+-----------------+
|            NULL |
+-----------------+
```

- `gen_rnd_email()`

  Generates a random email address in the `example.com` domain.

  Arguments:

  None.

  Return value:

  A random email address as a string.

  Example:

```
mysql> SELECT gen_rnd_email();
+--------------------------+
| gen_rnd_email()          |
+--------------------------+
| ijocv.mwvhhuf@example.com |
+--------------------------+
```

- `gen_rnd_pan([size])`

  Generates a random payment card Primary Account Number. The number passes the Luhn check (an algorithm that performs a checksum verification against a check digit).

  Warning

  Values returned from `gen_rnd_pan()` should be used only for test purposes, and are not suitable for publication. There is no way to guarantee that a given return value is not assigned to a legitimate payment account. Should it be necessary to publish a `gen_rnd_pan()` result, consider masking it with `mask_pan()` or `mask_pan_relaxed()`.

  Arguments:

  - *`size`*: (Optional) An integer that specifies the size of the result. The default is 16 if *`size`* is not given. If given, *`size`* must be an integer in the range from 12 to 19.

  Return value:

  A random payment number as a string, or `NULL` if a *`size`* argument outside the permitted range is given.

  Example:

```
mysql> SELECT mask_pan(gen_rnd_pan());
+-------------------------+
| mask_pan(gen_rnd_pan()) |
+-------------------------+
```

```
| XXXXXXXXXXX5805        |
+------------------------+
mysql> SELECT mask_pan(gen_rnd_pan(19));
+--------------------------+
| mask_pan(gen_rnd_pan(19)) |
+--------------------------+
| XXXXXXXXXXXXXX5067       |
+--------------------------+
mysql> SELECT mask_pan_relaxed(gen_rnd_pan());
+---------------------------------+
| mask_pan_relaxed(gen_rnd_pan()) |
+---------------------------------+
| 398403XXXXXX9547                |
+---------------------------------+
mysql> SELECT mask_pan_relaxed(gen_rnd_pan(19));
+----------------------------------+
| mask_pan_relaxed(gen_rnd_pan(19)) |
+----------------------------------+
| 578416XXXXXXXXX6509              |
+----------------------------------+
mysql> SELECT gen_rnd_pan(11), gen_rnd_pan(20);
+-----------------+-----------------+
| gen_rnd_pan(11) | gen_rnd_pan(20) |
+-----------------+-----------------+
| NULL            | NULL            |
+-----------------+-----------------+
```

- `gen_rnd_ssn()`

  Generates a random US Social Security number in *AAA-BB-CCCC* format. The *AAA* part is greater than 900 and the *BB* part is less than 70, which are characteristics not used for legitimate Social Security numbers.

  Arguments:

  None.

  Return value:

  A random Social Security number as a string.

  Example:

```
mysql> SELECT gen_rnd_ssn();
+---------------+
| gen_rnd_ssn() |
+---------------+
| 951-26-0058   |
+---------------+
```

- `gen_rnd_us_phone()`

  Generates a random US phone number in `1-555-`*AAA-BBBB* format. The 555 area code is not used for legitimate phone numbers.

  Arguments:

  None.

  Return value:

  A random US phone number as a string.

  Example:

```
mysql> SELECT gen_rnd_us_phone();
+-------------------+
| gen_rnd_us_phone() |
```

```
+--------------------+
| 1-555-682-5423     |
+--------------------+
```

Random Data Dictionary-Based Functions

The functions in this section manipulate dictionaries of terms and perform generation and masking operations based on them. Some of these functions require the `SUPER` privilege.

When a dictionary is loaded, it becomes part of the dictionary registry and is assigned a name to be used by other dictionary functions. Dictionaries are loaded from plain text files containing one term per line. Empty lines are ignored. To be valid, a dictionary file must contain at least one nonempty line.

- `gen_blacklist(`*`str`*`, `*`dictionary_name`*`, `*`replacement_dictionary_name`*`)`

  Replaces a term present in one dictionary with a term from a second dictionary and returns the replacement term. This masks the original term by substitution. This function is deprecated in MySQL 8.0.23; use `gen_blocklist()` instead.

- `gen_blocklist(`*`str`*`, `*`dictionary_name`*`, `*`replacement_dictionary_name`*`)`

  Replaces a term present in one dictionary with a term from a second dictionary and returns the replacement term. This masks the original term by substitution. This function was added in MySQL 8.0.23; use it instead of `gen_blacklist()`.

  Arguments:

  - *`str`*: A string that indicates the term to replace.
  - *`dictionary_name`*: A string that names the dictionary containing the term to replace.
  - *`replacement_dictionary_name`*: A string that names the dictionary from which to choose the replacement term.

  Return value:

  A string randomly chosen from *`replacement_dictionary_name`* as a replacement for *`str`*, or *`str`* if it does not appear in *`dictionary_name`*, or `NULL` if either dictionary name is not in the dictionary registry.

  If the term to replace appears in both dictionaries, it is possible for the return value to be the same term.

  Example:

```
mysql> SELECT gen_blocklist('Berlin', 'DE_Cities', 'US_Cities');
+-------------------------------------------------+
| gen_blocklist('Berlin', 'DE_Cities', 'US_Cities') |
+-------------------------------------------------+
| Phoenix                                         |
+-------------------------------------------------+
```

- `gen_dictionary(`*`dictionary_name`*`)`

  Returns a random term from a dictionary.

  Arguments:

  - *`dictionary_name`*: A string that names the dictionary from which to choose the term.

  Return value:

  A random term from the dictionary as a string, or `NULL` if the dictionary name is not in the dictionary registry.

Example:

```
mysql> SELECT gen_dictionary('mydict');
+-------------------------+
| gen_dictionary('mydict') |
+-------------------------+
| My term                  |
+-------------------------+
mysql> SELECT gen_dictionary('no-such-dict');
+-------------------------------+
| gen_dictionary('no-such-dict') |
+-------------------------------+
| NULL                          |
+-------------------------------+
```

- `gen_dictionary_drop(`*`dictionary_name`*`)`

  Removes a dictionary from the dictionary registry.

  This function requires the `SUPER` privilege.

  Arguments:

  - *`dictionary_name`*: A string that names the dictionary to remove from the dictionary registry.

  Return value:

  A string that indicates whether the drop operation succeeded. `Dictionary removed` indicates success. `Dictionary removal error` indicates failure.

  Example:

```
mysql> SELECT gen_dictionary_drop('mydict');
+-----------------------------+
| gen_dictionary_drop('mydict') |
+-----------------------------+
| Dictionary removed            |
+-----------------------------+
mysql> SELECT gen_dictionary_drop('no-such-dict');
+-----------------------------------+
| gen_dictionary_drop('no-such-dict') |
+-----------------------------------+
| Dictionary removal error          |
+-----------------------------------+
```

- `gen_dictionary_load(dictionary_path, dictionary_name)`

  Loads a file into the dictionary registry and assigns the dictionary a name to be used with other functions that require a dictionary name argument.

  This function requires the `SUPER` privilege.

  Important

  Dictionaries are not persistent. Any dictionary used by applications must be loaded for each server startup.

  Once loaded into the registry, a dictionary is used as is, even if the underlying dictionary file changes. To reload a dictionary, first drop it with `gen_dictionary_drop()`, then load it again with `gen_dictionary_load()`.

  Arguments:

  - *`dictionary_path`*: A string that specifies the path name of the dictionary file.
  - *`dictionary_name`*: A string that provides a name for the dictionary.

  Return value:

  A string that indicates whether the load operation succeeded. `Dictionary load success` indicates success. `Dictionary load error` indicates failure. Dictionary load failure can occur for several reasons, including:

  - A dictionary with the given name is already loaded.
  - The dictionary file is not found.
  - The dictionary file contains no terms.
  - The `secure_file_priv` system variable is set and the dictionary file is not located in the directory named by the variable.

  Example:

```
mysql> SELECT gen_dictionary_load('/usr/local/mysql/mysql-files/mydict','mydict');
+-----------------------------------------------------------------------+
| gen_dictionary_load('/usr/local/mysql/mysql-files/mydict','mydict') |
+-----------------------------------------------------------------------+
| Dictionary load success                                               |
+-----------------------------------------------------------------------+
mysql> SELECT gen_dictionary_load('/dev/null','null');
+---------------------------------------+
| gen_dictionary_load('/dev/null','null') |
+---------------------------------------+
| Dictionary load error                 |
+---------------------------------------+
```

6.6 MySQL Enterprise Encryption

Note

MySQL Enterprise Encryption is an extension included in MySQL Enterprise Edition, a commercial product. To learn more about commercial products, https://www.mysql.com/products/.

MySQL Enterprise Edition includes a set of encryption functions that expose OpenSSL capabilities at the SQL level. The functions enable Enterprise applications to perform the following operations:

- Implement added data protection using public-key asymmetric cryptography

- Create public and private keys and digital signatures
- Perform asymmetric encryption and decryption
- Use cryptographic hashing for digital signing and data verification and validation

In releases before MySQL 8.0.30, these functions are based on the `openssl_udf` shared library. From MySQL 8.0.30, they are provided by a MySQL component `component_enterprise_encryption`.

6.6.1 MySQL Enterprise Encryption Installation and Upgrading

In releases before MySQL 8.0.30, the functions provided by MySQL Enterprise Encryption are installed by creating them individually, based on the `openssl_udf` shared library. From MySQL 8.0.30, the functions are provided by a MySQL component `component_enterprise_encryption`, and installing the component installs all of the functions. The functions from the `openssl_udf` shared library are deprecated from that release, and you should upgrade to the component instead.

Installation From MySQL 8.0.30

From MySQL 8.0.30, MySQL Enterprise Encryption's functions are provided by a MySQL component `component_enterprise_encryption`, rather than being installed from the `openssl_udf` shared library. If you are upgrading to MySQL 8.0.30 from an earlier release where you used MySQL Enterprise Encryption, the functions you created remain available and are supported. However, these legacy functions are deprecated from this release, and it is recommended that you install the component instead. The component functions are backward compatible. For upgrade information, see Upgrading MySQL Enterprise Encryption.

If you are upgrading, before installing the component, unload the legacy functions using the `DROP FUNCTION` statement:

```
DROP FUNCTION asymmetric_decrypt;
DROP FUNCTION asymmetric_derive;
DROP FUNCTION asymmetric_encrypt;
DROP FUNCTION asymmetric_sign;
DROP FUNCTION asymmetric_verify;
DROP FUNCTION create_asymmetric_priv_key;
DROP FUNCTION create_asymmetric_pub_key;
DROP FUNCTION create_dh_parameters;
DROP FUNCTION create_digest;
```

The function names must be specified in lowercase. The statements require the `DROP` privilege for the `mysql` database.

To install the component, issue an `INSTALL COMPONENT` statement:

```
INSTALL COMPONENT "file://component_enterprise_encryption";
```

`INSTALL COMPONENT` requires the `INSERT` privilege for the `mysql.component` system table because it adds a row to that table to register the component. To verify that the component has been installed, issue:

```
SELECT * FROM mysql.component;
```

Components listed in `mysql.component` are loaded by the loader service during the startup sequence.

If you need to uninstall the component, issue an `UNINSTALL COMPONENT` statement:

```
UNINSTALL COMPONENT "file://component_enterprise_encryption";
```

For more details, see Section 5.5.1, “Installing and Uninstalling Components”.

Installing the component installs all of the functions, so you do not need to create them using `CREATE FUNCTION` statements as you do before MySQL 8.0.30. Uninstalling the component uninstalls all of the functions.

When you have installed the component, if you want the component functions to support decryption and verification for content produced by the legacy functions before MySQL 8.0.30, set the component’s system variable `enterprise_encryption.rsa_support_legacy_padding` to `ON`. Also, if you want to change the maximum length allowed for the RSA keys generated by the component functions, use the component’s system variable `enterprise_encryption.maximum_rsa_key_size` to set an appropriate maximum. For configuration information, see Section 6.6.2, “Configuring MySQL Enterprise Encryption”.

Installation To MySQL 8.0.29

Before MySQL 8.0.29, MySQL Enterprise Encryption functions are located in a loadable function library file installed in the plugin directory (the directory named by the `plugin_dir` system variable). The function library base name is `openssl_udf` and the suffix is platform dependent. For example, the file name on Linux or Windows is `openssl_udf.so` or `openssl_udf.dll`, respectively.

To install functions from the `openssl_udf` shared library file, use the `CREATE FUNCTION` statement. To load all functions from the library, use this set of statements, adjusting the file name suffix as necessary:

```
CREATE FUNCTION asymmetric_decrypt RETURNS STRING
  SONAME 'openssl_udf.so';
CREATE FUNCTION asymmetric_derive RETURNS STRING
  SONAME 'openssl_udf.so';
CREATE FUNCTION asymmetric_encrypt RETURNS STRING
  SONAME 'openssl_udf.so';
CREATE FUNCTION asymmetric_sign RETURNS STRING
  SONAME 'openssl_udf.so';
CREATE FUNCTION asymmetric_verify RETURNS INTEGER
  SONAME 'openssl_udf.so';
CREATE FUNCTION create_asymmetric_priv_key RETURNS STRING
  SONAME 'openssl_udf.so';
CREATE FUNCTION create_asymmetric_pub_key RETURNS STRING
  SONAME 'openssl_udf.so';
CREATE FUNCTION create_dh_parameters RETURNS STRING
  SONAME 'openssl_udf.so';
CREATE FUNCTION create_digest RETURNS STRING
  SONAME 'openssl_udf.so';
```

Once installed, the functions remain installed across server restarts. If you need to unload the functions, use the `DROP FUNCTION` statement:

```
DROP FUNCTION asymmetric_decrypt;
DROP FUNCTION asymmetric_derive;
DROP FUNCTION asymmetric_encrypt;
DROP FUNCTION asymmetric_sign;
DROP FUNCTION asymmetric_verify;
DROP FUNCTION create_asymmetric_priv_key;
DROP FUNCTION create_asymmetric_pub_key;
DROP FUNCTION create_dh_parameters;
DROP FUNCTION create_digest;
```

In the `CREATE FUNCTION` and `DROP FUNCTION` statements, the function names must be specified in lowercase. This differs from their use at function invocation time, for which you can use any lettercase.

The `CREATE FUNCTION` and `DROP FUNCTION` statements require the `INSERT` and `DROP` privilege, respectively, for the `mysql` database.

The functions provided by the `openssl_udf` shared library allow a minimum key size of 1024 bits. You can set a maximum key size using the `MYSQL_OPENSSL_UDF_RSA_BITS_THRESHOLD`, `MYSQL_OPENSSL_UDF_DSA_BITS_THRESHOLD`, and `MYSQL_OPENSSL_UDF_DH_BITS_THRESHOLD` environment variables, as described in Section 6.6.2, "Configuring MySQL Enterprise Encryption". If you do not set a maximum key size, the upper limit is 16384 for the RSA algorithm and 10000 for the DSA algorithm, as specified by OpenSSL.

Upgrading MySQL Enterprise Encryption

If you upgrade to MySQL 8.0.30 or later from an earlier release where you used the functions provided by the `openssl_udf` shared library, the functions you created remain available and are supported. However, these legacy functions are deprecated from MySQL 8.0.30, and it is recommended that you install the MySQL Enterprise Encryption component `component_enterprise_encryption` instead.

When you are upgrading, before installing the component, you must unload the legacy functions using the `DROP FUNCTION` statement. For instructions to do this, see Installation From MySQL 8.0.30.

The component functions are backward compatible:

- RSA public and private keys generated by the legacy functions can be used with the component functions.
- Data encrypted with the legacy functions can be decrypted by the component functions.
- Signatures created by the legacy functions can be verified with the component functions.

For the component functions to support decryption and verification for content produced by the legacy functions, you must set the system variable `enterprise_encryption.rsa_support_legacy_padding` to `ON` (the default is `OFF`). For configuration information, see Section 6.6.2, "Configuring MySQL Enterprise Encryption".

The legacy functions cannot handle encrypted data, public keys, and signatures created by the component functions, due to the differences in the padding and key format used by the component functions to meet the current standards.

The new functions provided by the `component_enterprise_encryption` component have some differences in behavior and support from the legacy functions provided by the `openssl_udf` shared library. The most important of these are as follows:

- The legacy functions support the older DSA algorithm and Diffie-Hellman key exchange method. The component functions use only the generally preferred RSA algorithm.
- For the legacy functions, the minimum RSA key size is less than current best practice. The component functions follow current best practice on minimum RSA key size.
- The legacy functions support only SHA2 for digests, and require digests for signatures. The component functions also support SHA3 for digests (provided that OpenSSL 1.1.1 is in use), and do not require digests for signatures, although they support them.
- The `asymmetric_encrypt()` legacy function supports encryption using private keys. The `asymmetric_encrypt()` component function only accepts a public key. It is recommended that you only encrypt using public keys with the legacy function as well.
- The `create_dh_parameters()` and `asymmetric_derive()` legacy functions for the Diffie-Hellman key exchange method are not provided by the `component_enterprise_encryption` component.

Table 1 summarizes the technical differences in support and operation between the legacy functions provided by the `openssl_udf` shared library, and the functions provided by the `component_enterprise_encryption` component from MySQL 8.0.30.

Table 6.46 MySQL Enterprise Encryption functions

Capability	Legacy functions (to MySQL 8.0.29)	Component functions (from MySQL 8.0.30)
Encryption method	RSA, DSA, Diffie-Hellman (DH)	RSA only
Key for encryption	Private or public	Public only
RSA key format	PKCS #1 v1.5	PKCS #8
Minimum RSA key size	1024 bits	2048 bits
Maximum RSA key size limit	Set with environment variable MYSQL_OPENSSL_UDF_RSA_BITS_THRESHOLD, default limit is algorithm maximum 16384	Set with system variable component_enterprise_encryption.maximum_r default limit is 4096
Digest algorithms	SHA2	SHA2, SHA3 (with OpenSSL 1.1.1)
Signatures	Digest required	Digests supported but not required, any string of arbitrary length can be used
Output padding	RSAES-PKCS1-v1_5	RSAES-OAEP
Signature padding	RSASSA-PKCS1-v1_5	RSASSA-PSS

6.6.2 Configuring MySQL Enterprise Encryption

MySQL Enterprise Encryption lets you limit keys to a length that provides adequate security for your requirements while balancing this with resource usage. You can also configure the functions provided by the `component_enterprise_encryption` component from MySQL 8.0.30, to support decryption and verification for content produced by the legacy `openssl_udf` shared library functions.

Decryption Support By Component Functions For Legacy Functions

By default, the functions provided by the `component_enterprise_encryption` component from MySQL 8.0.30 do not decrypt encrypted text, or verify signatures, that were produced by the legacy functions provided in earlier releases by the `openssl_udf` shared library. The component functions assume that encrypted text uses the RSAES-OAEP padding scheme, and signatures use the RSASSA-PSS signature scheme. However, encrypted text produced by the legacy functions uses the RSAES-PKCS1-v1_5 padding scheme, and signatures produced by the legacy functions use the RSASSA-PKCS1-v1_5 signature scheme.

If you want the component functions to support content produced by the legacy functions before MySQL 8.0.30, set the component's system variable `enterprise_encryption.rsa_support_legacy_padding` to `ON`. The system variable is available when the component is installed. When you set it to `ON`, the component functions first attempt to decrypt or verify content assuming it has their normal schemes. If that does not work, they also attempt to decrypt or verify the content assuming it has the schemes used by the legacy functions. This behavior is not the default because it increases the time taken to process content that cannot be decrypted or verified at all. If you are not handling content produced by the legacy functions, leave the system variable to default to `OFF`.

Key Length Limits

The amount of CPU resources required by MySQL Enterprise Encryption's key generation functions increases as the key length increases. For some installations, this might result in unacceptable CPU usage if applications frequently generate excessively long keys.

OpenSSL specifies a minimum key length of 1024 bits for all keys. OpenSSL also specifies a maximum key length of 16384 bits for RSA keys, 10000 bits for DSA keys, and 10000 bits for DH keys.

From MySQL 8.0.30, the functions provided by the `component_enterprise_encryption` component have a higher minimum key length of 2048 bits for RSA keys, which is in line with current best practice for minimum key lengths. The component's system variable `enterprise_encryption.maximum_rsa_key_size` specifies the maximum key size, and it defaults to 4096 bits. You can change this to allow keys up to the maximum length allowed by OpenSSL, 16384 bits.

For releases before MySQL 8.0.30, the legacy functions provided by the `openssl_udf` shared library default to OpenSSL's minimum and maximum limits. If the maximum values are too high, you can specify a lower maximum key length using the following system variables:

- `MYSQL_OPENSSL_UDF_DSA_BITS_THRESHOLD`: Maximum DSA key length in bits for `create_asymmetric_priv_key()`. The minimum and maximum values for this variable are 1024 and 10000.
- `MYSQL_OPENSSL_UDF_RSA_BITS_THRESHOLD`: Maximum RSA key length in bits for `create_asymmetric_priv_key()`. The minimum and maximum values for this variable are 1024 and 16384.
- `MYSQL_OPENSSL_UDF_DH_BITS_THRESHOLD`: Maximum key length in bits for `create_dh_parameters()`. The minimum and maximum values for this variable are 1024 and 10000.

To use any of these environment variables, set them in the environment of the process that starts the server. If set, their values take precedence over the maximum key lengths imposed by OpenSSL. For example, to set a maximum key length of 4096 bits for DSA and RSA keys for `create_asymmetric_priv_key()`, set these variables:

```
export MYSQL_OPENSSL_UDF_DSA_BITS_THRESHOLD=4096
export MYSQL_OPENSSL_UDF_RSA_BITS_THRESHOLD=4096
```

The example uses Bourne shell syntax. The syntax for other shells may differ.

6.6.3 MySQL Enterprise Encryption Usage and Examples

To use MySQL Enterprise Encryption in applications, invoke the functions that are appropriate for the operations you wish to perform. This section demonstrates how to carry out some representative tasks.

In releases before MySQL 8.0.30, MySQL Enterprise Encryption's functions are based on the `openssl_udf` shared library. From MySQL 8.0.30, the functions are provided by a MySQL component `component_enterprise_encryption`. In some cases, the behavior of the component functions differs from the behavior of the legacy functions provided by the `openssl_udf`. For a list of the differences, see Upgrading MySQL Enterprise Encryption. For full details of the behavior of each component's functions, see Section 6.6.4, “MySQL Enterprise Encryption Function Reference”.

If you install the legacy functions then upgrade to MySQL 8.0.30 or later, the functions you created remain available, are supported, and continue to work in the same way. However, they are deprecated from MySQL 8.0.30, and it is recommended that you install the MySQL Enterprise Encryption component `component_enterprise_encryption` instead. For instructions to upgrade, see Installation From MySQL 8.0.30.

The following general considerations apply when choosing key lengths and encryption algorithms:

- The strength of encryption for private and public keys increases with the key size, but the time for key generation increases as well.
- For the legacy functions, generation of DH keys takes much longer than RSA or DSA keys. The component functions from MySQL 8.0.30 only support RSA keys.
- Asymmetric encryption functions consume more resources compared to symmetric functions. They are good for encrypting small amounts of data and creating and verifying signatures. For encrypting

large amounts of data, symmetric encryption functions are faster. MySQL Server provides the `AES_ENCRYPT()` and `AES_DECRYPT()` functions for symmetric encryption.

Key string values can be created at runtime and stored into a variable or table using `SET`, `SELECT`, or `INSERT`. This example works with both the component function and the legacy function:

```
SET @priv1 = create_asymmetric_priv_key('RSA', 2048);
SELECT create_asymmetric_priv_key('RSA', 2048) INTO @priv2;
INSERT INTO t (key_col) VALUES(create_asymmetric_priv_key('RSA', 1024));
```

Key string values stored in files can be read using the `LOAD_FILE()` function by users who have the `FILE` privilege. Digest and signature strings can be handled similarly.

- Create a private/public key pair
- Use the public key to encrypt data and the private key to decrypt it
- Generate a digest from a string
- Use the digest with a key pair

Create a private/public key pair

This example works with both the component functions and the legacy functions:

```
-- Encryption algorithm
SET @algo = 'RSA';
-- Key length in bits; make larger for stronger keys
SET @key_len = 2048;

-- Create private key
SET @priv = create_asymmetric_priv_key(@algo, @key_len);
-- Derive corresponding public key from private key, using same algorithm
SET @pub = create_asymmetric_pub_key(@algo, @priv);
```

You can use the key pair to encrypt and decrypt data or to sign and verify data.

Use the public key to encrypt data and the private key to decrypt it

This example works with both the component functions and the legacy functions. In both cases, the members of the key pair must be RSA keys:

```
SET @ciphertext = asymmetric_encrypt(@algo, 'My secret text', @pub);
SET @plaintext = asymmetric_decrypt(@algo, @ciphertext, @priv);
```

Generate a digest from a string

This example works with both the component functions and the legacy functions:

```
-- Digest type
SET @dig_type = 'SHA512';

-- Generate digest string
SET @dig = create_digest(@dig_type, 'My text to digest');
```

Use the digest with a key pair

The key pair can be used to sign data, then verify that the signature matches the digest. This example works with both the component functions and the legacy functions:

```
-- Encryption algorithm; keys must
-- have been created using same algorithm
SET @algo = 'RSA';
```

```
-- Digest algorithm to sign the data
SET @dig_type = 'SHA512';

-- Generate signature for digest and verify signature against digest
SET @sig = asymmetric_sign(@algo, @dig, @priv, @dig_type);
-- Verify signature against digest
SET @verf = asymmetric_verify(@algo, @dig, @sig, @pub, @dig_type);
```

For the legacy functions, signatures require a digest. For the component functions, signatures do not require a digest, and can use any data string. The digest type in these functions refers to the algorithm that is used to sign the data, not the algorithm that was used to create the original input for the signature. This example is for the component functions:

```
-- Encryption algorithm; keys must
-- have been created using same algorithm
SET @algo = 'RSA';
-- Arbitrary text string for signature
SET @text = repeat('j', 256);
-- Digest algorithm to sign the data
SET @dig_type = 'SHA512';

-- Generate signature for digest and verify signature against digest
SET @sig = asymmetric_sign(@algo, @text, @priv, @dig_type);
-- Verify signature against digest
SET @verf = asymmetric_verify(@algo, @text, @sig, @pub, @dig_type);
```

6.6.4 MySQL Enterprise Encryption Function Reference

In releases from MySQL 8.0.30, MySQL Enterprise Encryption's functions are provided by the MySQL component `component_enterprise_encryption`. For their descriptions, see Section 6.6.5, “MySQL Enterprise Encryption Component Function Descriptions”.

In releases before MySQL 8.0.30, MySQL Enterprise Encryption's functions are based on the `openssl_udf` shared library. The functions continue to be available in later releases if they have been installed, but they are deprecated. For their descriptions, see Section 6.6.6, “MySQL Enterprise Encryption Legacy Function Descriptions”.

For information on upgrading to the new component functions provided by the MySQL component `component_enterprise_encryption`, and a list of the behavior differences between the legacy functions and the component functions, see Upgrading MySQL Enterprise Encryption.

6.6.5 MySQL Enterprise Encryption Component Function Descriptions

In releases from MySQL 8.0.30, MySQL Enterprise Encryption's functions are provided by the MySQL component `component_enterprise_encryption`. This reference describes those functions.

For information on upgrading to the new component functions provided by the MySQL component `component_enterprise_encryption`, and a list of the behavior differences between the legacy functions and the component functions, see Upgrading MySQL Enterprise Encryption.

The reference for the legacy functions in releases before MySQL 8.0.30 based on the `openssl_udf` shared library is Section 6.6.6, “MySQL Enterprise Encryption Legacy Function Descriptions”.

MySQL Enterprise Encryption functions have these general characteristics:

- For arguments of the wrong type or an incorrect number of arguments, each function returns an error.
- If the arguments are not suitable to permit a function to perform the requested operation, it returns `NULL` or 0 as appropriate. This occurs, for example, if a function does not support a specified algorithm, a key length is too short or long, or a string expected to be a key string in PEM format is not a valid key.

- The underlying SSL library takes care of randomness initialization.

The component functions only support the RSA encryption algorithm.

For additional examples and discussion, see Section 6.6.3, “MySQL Enterprise Encryption Usage and Examples”.

- `asymmetric_decrypt(algorithm, data_str, priv_key_str)`

  Decrypts an encrypted string using the given algorithm and key string, and returns the resulting plaintext as a binary string. If decryption fails, the result is `NULL`.

  For the legacy version of this function in use before MySQL 8.0.29, see Section 6.6.6, “MySQL Enterprise Encryption Legacy Function Descriptions”.

  By default, the `component_enterprise_encryption` function assumes that encrypted text uses the RSAES-OAEP padding scheme. The function supports decryption for content encrypted by the legacy `openssl_udf` shared library functions if the system variable `enterprise_encryption.rsa_support_legacy_padding` is set to `ON` (the default is `OFF`). When `ON` is set, the function also supports the RSAES-PKCS1-v1_5 padding scheme, as used by the legacy `openssl_udf` shared library functions. When `OFF` is set, content encrypted by the legacy functions cannot be decrypted, and the function returns null output for such content.

  `algorithm` is the encryption algorithm used to create the key. The supported algorithm value is `'RSA'`.

  `data_str` is the encrypted string to decrypt, which was encrypted with `asymmetric_encrypt()`.

  `priv_key_str` is a valid PEM encoded RSA private key. For successful decryption, the key string must correspond to the public key string used with `asymmetric_encrypt()` to produce the encrypted string. The `asymmetric_encrypt()` component function only supports encryption using a public key, so decryption takes place with the corresponding private key.

  For a usage example, see the description of `asymmetric_encrypt()`.

- `asymmetric_encrypt(algorithm, data_str, pub_key_str)`

  Encrypts a string using the given algorithm and key string, and returns the resulting ciphertext as a binary string. If encryption fails, the result is `NULL`.

  For the legacy version of this function in use before MySQL 8.0.29, see Section 6.6.6, “MySQL Enterprise Encryption Legacy Function Descriptions”.

  `algorithm` is the encryption algorithm used to create the key. The supported algorithm value is `'RSA'`.

  `data_str` is the string to encrypt. The length of this string cannot be greater than the key string length in bytes, minus 42 (to account for the padding).

  `pub_key_str` is a valid PEM encoded RSA public key. The `asymmetric_encrypt()` component function only supports encryption using a public key.

  To recover the original unencrypted string, pass the encrypted string to `asymmetric_decrypt()`, along with the other part of the key pair used for encryption, as in the following example:

```
-- Generate private/public key pair
SET @priv = create_asymmetric_priv_key('RSA', 2048);
SET @pub = create_asymmetric_pub_key('RSA', @priv);

-- Encrypt using public key, decrypt using private key
SET @ciphertext = asymmetric_encrypt('RSA', 'The quick brown fox', @pub);
SET @plaintext = asymmetric_decrypt('RSA', @ciphertext, @priv);
```

Suppose that:

```
SET @s = a string to be encrypted
SET @priv = a valid private RSA key string in PEM format
SET @pub = the corresponding public RSA key string in PEM format
```

Then these identity relationships hold:

```
asymmetric_decrypt('RSA', asymmetric_encrypt('RSA', @s, @pub), @priv) = @s
```

- `asymmetric_sign(algorithm, text, priv_key_str, digest_type)`

  Signs a digest string or data string using a private key, and returns the signature as a binary string. If signing fails, the result is `NULL`.

  For the legacy version of this function in use before MySQL 8.0.29, see Section 6.6.6, “MySQL Enterprise Encryption Legacy Function Descriptions”.

  `algorithm` is the encryption algorithm used to create the key. The supported algorithm value is `'RSA'`.

  `text` is a data string or digest string. The function accepts digests but does not require them, as it is also capable of handling data strings of an arbitrary length. A digest string can be generated by calling `create_digest()`.

  `priv_key_str` is the private key string to use for signing the digest string. It must be a valid PEM encoded RSA private key.

  `digest_type` is the algorithm to be used to sign the data. The supported *`digest_type`* values are `'SHA224'`, `'SHA256'`, `'SHA384'`, and `'SHA512'` when OpenSSL 1.0.1 is in use. If OpenSSL 1.1.1 is in use, the additional *`digest_type`* values `'SHA3-224'`, `'SHA3-256'`, `'SHA3-384'`, and `'SHA3-512'` are available.

  For a usage example, see the description of `asymmetric_verify()`.

- `asymmetric_verify(algorithm, text, sig_str, pub_key_str, digest_type)`

  Verifies whether the signature string matches the digest string, and returns 1 or 0 to indicate whether verification succeeded or failed. If verification fails, the result is `NULL`.

  For the legacy version of this function in use before MySQL 8.0.29, see Section 6.6.6, “MySQL Enterprise Encryption Legacy Function Descriptions”.

  By default, the `component_enterprise_encryption` function assumes that signatures use the RSASSA-PSS signature scheme. The function supports verification for signatures produced by the legacy `openssl_udf` shared library functions if the system variable `enterprise_encryption.rsa_support_legacy_padding` is set to `ON` (the default is `OFF`). When `ON` is set, the function also supports the RSASSA-PKCS1-v1_5 signature scheme, as used

by the legacy `openssl_udf` shared library functions. When `OFF` is set, signatures produced by the legacy functions cannot be verified, and the function returns null output for such content.

`algorithm` is the encryption algorithm used to create the key. The supported algorithm value is `'RSA'`.

`text` is a data string or digest string. The component function accepts digests but does not require them, as it is also capable of handling data strings of an arbitrary length. A digest string can be generated by calling `create_digest()`.

`sig_str` is the signature string to be verified. A signature string can be generated by calling `asymmetric_sign()`.

`pub_key_str` is the public key string of the signer. It corresponds to the private key passed to `asymmetric_sign()` to generate the signature string. It must be a valid PEM encoded RSA public key.

`digest_type` is the algorithm that was used to sign the data. The supported *`digest_type`* values are `'SHA224'`, `'SHA256'`, `'SHA384'`, and `'SHA512'` when OpenSSL 1.0.1 is in use. If OpenSSL 1.1.1 is in use, the additional *`digest_type`* values `'SHA3-224'`, `'SHA3-256'`, `'SHA3-384'`, and `'SHA3-512'` are available.

```
-- Set the encryption algorithm and digest type
SET @algo = 'RSA';
SET @dig_type = 'SHA512';

-- Create private/public key pair
SET @priv = create_asymmetric_priv_key(@algo, 2048);
SET @pub = create_asymmetric_pub_key(@algo, @priv);

-- Generate digest from string
SET @dig = create_digest(@dig_type, 'The quick brown fox');

-- Generate signature for digest and verify signature against digest
SET @sig = asymmetric_sign(@algo, @dig, @priv, @dig_type);
SET @verf = asymmetric_verify(@algo, @dig, @sig, @pub, @dig_type);
```

- `create_asymmetric_priv_key(`*`algorithm, key_length`*`)`

  Creates a private key using the given algorithm and key length, and returns the key as a binary string in PEM format. The key is in PKCS #8 format. If key generation fails, the result is `NULL`.

  For the legacy version of this function in use before MySQL 8.0.29, see Section 6.6.6, “MySQL Enterprise Encryption Legacy Function Descriptions”.

  `algorithm` is the encryption algorithm used to create the key. The supported algorithm value is `'RSA'`.

  `key_length` is the key length in bits. If you exceed the maximum allowed key length or specify less than the minimum, key generation fails and the result is null output. The minimum allowed key length in bits is 2048. The maximum allowed key length is the value of the `enterprise_encryption.maximum_rsa_key_size` system variable, which defaults to 4096. It has a maximum setting of 16384, which is the maximum key length allowed for the RSA algorithm. See Section 6.6.2, “Configuring MySQL Enterprise Encryption”.

  Note

  Generating longer keys can consume significant CPU resources. Limiting the key length using the `enterprise_encryption.maximum_rsa_key_size`

system variable lets you provide adequate security for your requirements while balancing this with resource usage.

This example creates a 2048-bit RSA private key, then derives a public key from the private key:

```
SET @priv = create_asymmetric_priv_key('RSA', 2048);
SET @pub = create_asymmetric_pub_key('RSA', @priv);
```

- `create_asymmetric_pub_key(algorithm, priv_key_str)`

  Derives a public key from the given private key using the given algorithm, and returns the key as a binary string in PEM format. The key is in PKCS #8 format. If key derivation fails, the result is `NULL`.

  For the legacy version of this function in use before MySQL 8.0.29, see Section 6.6.6, “MySQL Enterprise Encryption Legacy Function Descriptions”.

  `algorithm` is the encryption algorithm used to create the key. The supported algorithm value is `'RSA'`.

  `priv_key_str` is a valid PEM encoded RSA private key.

  For a usage example, see the description of `create_asymmetric_priv_key()`.

- `create_digest(digest_type, str)`

  Creates a digest from the given string using the given digest type, and returns the digest as a binary string. If digest generation fails, the result is `NULL`.

  For the legacy version of this function in use before MySQL 8.0.29, see Section 6.6.6, “MySQL Enterprise Encryption Legacy Function Descriptions”.

  The resulting digest string is suitable for use with `asymmetric_sign()` and `asymmetric_verify()`. The component versions of these functions accept digests but do not require them, as they are capable of handling data of an arbitrary length.

  `digest_type` is the digest algorithm to be used to generate the digest string. The supported *`digest_type`* values are `'SHA224'`, `'SHA256'`, `'SHA384'`, and `'SHA512'` when OpenSSL 1.0.1 is in use. If OpenSSL 1.1.1 is in use, the additional *`digest_type`* values `'SHA3-224'`, `'SHA3-256'`, `'SHA3-384'`, and `'SHA3-512'` are available.

  `str` is the non-null data string for which the digest is to be generated.

```
SET @dig = create_digest('SHA512', 'The quick brown fox');
```

6.6.6 MySQL Enterprise Encryption Legacy Function Descriptions

In releases before MySQL 8.0.30, MySQL Enterprise Encryption's functions are based on the `openssl_udf` shared library. This reference describes those functions. The functions continue to be available in later releases if they have been installed, but they are deprecated.

For information on upgrading to the new component functions provided by the MySQL component `component_enterprise_encryption`, and a list of the behavior differences between the legacy functions and the component functions, see Upgrading MySQL Enterprise Encryption.

The reference for the component functions is Section 6.6.5, “MySQL Enterprise Encryption Component Function Descriptions”.

MySQL Enterprise Encryption functions have these general characteristics:

- For arguments of the wrong type or an incorrect number of arguments, each function returns an error.

- If the arguments are not suitable to permit a function to perform the requested operation, it returns `NULL` or 0 as appropriate. This occurs, for example, if a function does not support a specified algorithm, a key length is too short or long, or a string expected to be a key string in PEM format is not a valid key.
- The underlying SSL library takes care of randomness initialization.

Several of the legacy functions take an encryption algorithm argument. The following table summarizes the supported algorithms by function.

Table 6.47 Supported Algorithms by Function

Function	Supported Algorithms
`asymmetric_decrypt()`	RSA
`asymmetric_derive()`	DH
`asymmetric_encrypt()`	RSA
`asymmetric_sign()`	RSA, DSA
`asymmetric_verify()`	RSA, DSA
`create_asymmetric_priv_key()`	RSA, DSA, DH
`create_asymmetric_pub_key()`	RSA, DSA, DH
`create_dh_parameters()`	DH

Note

Although you can create keys using any of the RSA, DSA, or DH encryption algorithms, other legacy functions that take key arguments might accept only certain types of keys. For example, `asymmetric_encrypt()` and `asymmetric_decrypt()` accept only RSA keys.

For additional examples and discussion, see Section 6.6.3, “MySQL Enterprise Encryption Usage and Examples”.

- `asymmetric_decrypt(`*`algorithm`*`, `*`crypt_str`*`, `*`key_str`*`)`

  Decrypts an encrypted string using the given algorithm and key string, and returns the resulting plaintext as a binary string. If decryption fails, the result is `NULL`.

  The `openssl_udf` shared library function cannot decrypt content produced by the `component_enterprise_encryption` functions that are available from MySQL 8.0.30.

  `algorithm` is the encryption algorithm used to create the key. The supported algorithm value is `'RSA'`.

  `crypt_str` is the encrypted string to decrypt, which was encrypted with `asymmetric_encrypt()`.

  `key_str` is a valid PEM encoded RSA public or private key. For successful decryption, the key string must correspond to the public or private key string used with `asymmetric_encrypt()` to produce the encrypted string.

  For a usage example, see the description of `asymmetric_encrypt()`.

- `asymmetric_derive(`*`pub_key_str`*`, `*`priv_key_str`*`)`

  Derives a symmetric key using the private key of one party and the public key of another, and returns the resulting key as a binary string. If key derivation fails, the result is `NULL`.

  `pub_key_str` and *`priv_key_str`* are valid PEM encoded key strings that were created using the DH algorithm.

Suppose that you have two pairs of public and private keys:

```
SET @dhp = create_dh_parameters(1024);
SET @priv1 = create_asymmetric_priv_key('DH', @dhp);
SET @pub1 = create_asymmetric_pub_key('DH', @priv1);
SET @priv2 = create_asymmetric_priv_key('DH', @dhp);
SET @pub2 = create_asymmetric_pub_key('DH', @priv2);
```

Suppose further that you use the private key from one pair and the public key from the other pair to create a symmetric key string. Then this symmetric key identity relationship holds:

```
asymmetric_derive(@pub1, @priv2) = asymmetric_derive(@pub2, @priv1)
```

This example requires DH private/public keys as inputs, created using a shared symmetric secret. Create the secret by passing the key length to `create_dh_parameters()`, then pass the secret as the “key length” to `create_asymmetric_priv_key()`.

```
-- Generate DH shared symmetric secret
SET @dhp = create_dh_parameters(1024);
-- Generate DH key pairs
SET @algo = 'DH';
SET @priv1 = create_asymmetric_priv_key(@algo, @dhp);
SET @pub1 = create_asymmetric_pub_key(@algo, @priv1);
SET @priv2 = create_asymmetric_priv_key(@algo, @dhp);
SET @pub2 = create_asymmetric_pub_key(@algo, @priv2);

-- Generate symmetric key using public key of first party,
-- private key of second party
SET @sym1 = asymmetric_derive(@pub1, @priv2);

-- Or use public key of second party, private key of first party
SET @sym2 = asymmetric_derive(@pub2, @priv1);
```

- `asymmetric_encrypt(`*`algorithm, str, key_str`*`)`

  Encrypts a string using the given algorithm and key string, and returns the resulting ciphertext as a binary string. If encryption fails, the result is `NULL`.

  `algorithm` is the encryption algorithm used to create the key. The supported algorithm value is `'RSA'`.

  `str` is the string to encrypt. The length of this string cannot be greater than the key string length in bytes, minus 11 (to account for the padding).

  `key_str` is a valid PEM encoded RSA public or private key.

  To recover the original unencrypted string, pass the encrypted string to `asymmetric_decrypt()`, along with the other part of the key pair used for encryption, as in the following example:

  ```
  -- Generate private/public key pair
  SET @priv = create_asymmetric_priv_key('RSA', 1024);
  SET @pub = create_asymmetric_pub_key('RSA', @priv);

  -- Encrypt using private key, decrypt using public key
  SET @ciphertext = asymmetric_encrypt('RSA', 'The quick brown fox', @priv);
  SET @plaintext = asymmetric_decrypt('RSA', @ciphertext, @pub);

  -- Encrypt using public key, decrypt using private key
  SET @ciphertext = asymmetric_encrypt('RSA', 'The quick brown fox', @pub);
  SET @plaintext = asymmetric_decrypt('RSA', @ciphertext, @priv);
  ```

  Suppose that:

  ```
  SET @s = a string to be encrypted
  SET @priv = a valid private RSA key string in PEM format
  ```

```
SET @pub = the corresponding public RSA key string in PEM format
```

Then these identity relationships hold:

```
asymmetric_decrypt('RSA', asymmetric_encrypt('RSA', @s, @priv), @pub) = @s
asymmetric_decrypt('RSA', asymmetric_encrypt('RSA', @s, @pub), @priv) = @s
```

- `asymmetric_sign(algorithm, digest_str, priv_key_str, digest_type)`

  Signs a digest string using a private key string, and returns the signature as a binary string. If signing fails, the result is `NULL`.

  `algorithm` is the encryption algorithm used to create the key. The supported algorithm values are `'RSA'` and `'DSA'`.

  `digest_str` is a digest string. A digest string can be generated by calling `create_digest()`.

  `priv_key_str` is the private key string to use for signing the digest string. It can be a valid PEM encoded RSA private key or DSA private key.

  `digest_type` is the algorithm to be used to sign the data. The supported *`digest_type`* values are `'SHA224'`, `'SHA256'`, `'SHA384'`, and `'SHA512'`.

  For a usage example, see the description of `asymmetric_verify()`.

- `asymmetric_verify(algorithm, digest_str, sig_str, pub_key_str, digest_type)`

  Verifies whether the signature string matches the digest string, and returns 1 or 0 to indicate whether verification succeeded or failed. If verification fails, the result is `NULL`.

  The `openssl_udf` shared library function cannot verify content produced by the `component_enterprise_encryption` functions that are available from MySQL 8.0.30.

  `algorithm` is the encryption algorithm used to create the key. The supported algorithm values are `'RSA'` and `'DSA'`.

  `digest_str` is the digest string. A digest string is required, and can be generated by calling `create_digest()`.

  `sig_str` is the signature string to be verified. A signature string can be generated by calling `asymmetric_sign()`.

  `pub_key_str` is the public key string of the signer. It corresponds to the private key passed to `asymmetric_sign()` to generate the signature string. It must be a valid PEM encoded RSA public key or DSA public key.

  `digest_type` is the algorithm that was used to sign the data. The supported *`digest_type`* values are `'SHA224'`, `'SHA256'`, `'SHA384'`, and `'SHA512'`.

```
-- Set the encryption algorithm and digest type
SET @algo = 'RSA';
SET @dig_type = 'SHA224';

-- Create private/public key pair
SET @priv = create_asymmetric_priv_key(@algo, 1024);
SET @pub = create_asymmetric_pub_key(@algo, @priv);

-- Generate digest from string
SET @dig = create_digest(@dig_type, 'The quick brown fox');

-- Generate signature for digest and verify signature against digest
SET @sig = asymmetric_sign(@algo, @dig, @priv, @dig_type);
SET @verf = asymmetric_verify(@algo, @dig, @sig, @pub, @dig_type);
```

- `create_asymmetric_priv_key(algorithm, {key_len|dh_secret})`

  Creates a private key using the given algorithm and key length or DH secret, and returns the key as a binary string in PEM format. The key is in PKCS #1 format. If key generation fails, the result is `NULL`.

  `algorithm` is the encryption algorithm used to create the key. The supported algorithm values are `'RSA'`, `'DSA'`, and `'DH'`.

  `key_len` is the key length in bits for RSA and DSA keys. If you exceed the maximum allowed key length or specify less than the minimum, key generation fails and the result is null output. The minimum allowed key length in bits is 1,024, and the maximum allowed key length is 16,384 for the RSA algorithm or 10,000 for the DSA algorithm. These key-length limits are constraints imposed by OpenSSL. Server administrators can impose additional limits on maximum key length by setting the `MYSQL_OPENSSL_UDF_RSA_BITS_THRESHOLD`, `MYSQL_OPENSSL_UDF_DSA_BITS_THRESHOLD`, and `MYSQL_OPENSSL_UDF_DH_BITS_THRESHOLD` environment variables. See Section 6.6.2, “Configuring MySQL Enterprise Encryption”.

  Note

  Generating longer keys can consume significant CPU resources. Limiting the key length using the environment variables lets you provide adequate security for your requirements while balancing this with resource usage.

  `dh_secret` is a shared DH secret, which must be passed instead of a key length for DH keys. To create the secret, pass the key length to `create_dh_parameters()`.

  This example creates a 2,048-bit DSA private key, then derives a public key from the private key:

  ```
  SET @priv = create_asymmetric_priv_key('DSA', 2048);
  SET @pub = create_asymmetric_pub_key('DSA', @priv);
  ```

  For an example showing DH key generation, see the description of `asymmetric_derive()`.

- `create_asymmetric_pub_key(algorithm, priv_key_str)`

  Derives a public key from the given private key using the given algorithm, and returns the key as a binary string in PEM format. The key is in PKCS #1 format. If key derivation fails, the result is `NULL`.

  `algorithm` is the encryption algorithm used to create the key. The supported algorithm values are `'RSA'`, `'DSA'`, and `'DH'`.

  `priv_key_str` is a valid PEM encoded RSA, DSA, or DH private key.

  For a usage example, see the description of `create_asymmetric_priv_key()`.

- `create_dh_parameters(key_len)`

  Creates a shared secret for generating a DH private/public key pair and returns a binary string that can be passed to `create_asymmetric_priv_key()`. If secret generation fails, the result is `NULL`.

  `key_len` is the key length. The minimum and maximum key lengths in bits are 1,024 and 10,000. These key-length limits are constraints imposed by OpenSSL. Server administrators can impose additional limits on maximum key length by setting the `MYSQL_OPENSSL_UDF_RSA_BITS_THRESHOLD`, `MYSQL_OPENSSL_UDF_DSA_BITS_THRESHOLD`, and `MYSQL_OPENSSL_UDF_DH_BITS_THRESHOLD` environment variables. See Section 6.6.2, “Configuring MySQL Enterprise Encryption”.

  For an example showing how to use the return value for generating symmetric keys, see the description of `asymmetric_derive()`.

  ```
  SET @dhp = create_dh_parameters(1024);
  ```

- `create_digest(digest_type, str)`

  Creates a digest from the given string using the given digest type, and returns the digest as a binary string. If digest generation fails, the result is `NULL`.

  The resulting digest string is suitable for use with `asymmetric_sign()` and `asymmetric_verify()`. A digest is required for these functions.

  `digest_type` is the digest algorithm to be used to generate the digest string. The supported *`digest_type`* values are `'SHA224'`, `'SHA256'`, `'SHA384'`, and `'SHA512'`.

  `str` is the non-null data string for which the digest is to be generated.

```
SET @dig = create_digest('SHA512', 'The quick brown fox');
```

6.7 SELinux

Security-Enhanced Linux (SELinux) is a mandatory access control (MAC) system that implements access rights by applying a security label referred to as an *SELinux context* to each system object. SELinux policy modules use SELinux contexts to define rules for how processes, files, ports, and other system objects interact with each other. Interaction between system objects is only permitted if a policy rule allows it.

An SELinux context (the label applied to a system object) has the following fields: `user`, `role`, `type`, and `security level`. Type information rather than the entire SELinux context is used most commonly to define rules for how processes interact with other system objects. MySQL SELinux policy modules, for example, define policy rules using `type` information.

You can view SELinux contexts using operating system commands such as `ls` and `ps` with the `-Z` option. Assuming that SELinux is enabled and a MySQL Server is running, the following commands show the SELinux context for the `mysqld` process and MySQL data directory:

`mysqld` process:

```
$> ps -eZ | grep mysqld
system_u:system_r:mysqld_t:s0    5924 ?        00:00:03 mysqld
```

MySQL data directory:

```
$> cd /var/lib
$> ls -Z | grep mysql
system_u:object_r:mysqld_db_t:s0 mysql
```

where:

- `system_u` is an SELinux user identity for system processes and objects.
- `system_r` is an SELinux role used for system processes.
- `objects_r` is an SELinux role used for system objects.
- `mysqld_t` is the type associated with the mysqld process.
- `mysqld_db_t` is the type associated with the MySQL data directory and its files.
- `s0` is the security level.

For more information about interpreting SELinux contexts, refer to your distribution's SELinux documentation.

6.7.1 Check if SELinux is Enabled

SELinux is enabled by default on some Linux distributions including Oracle Linux, RHEL, CentOS, and Fedora. Use the `sestatus` command to determine if SELinux is enabled on your distribution:

```
$> sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   enforcing
Mode from config file:          enforcing
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Memory protection checking:     actual (secure)
Max kernel policy version:      31
```

If SELinux is disabled or the `sestatus` command is not found, refer to your distribution's SELinux documentation for guidance before enabling SELinux.

6.7.2 Changing the SELinux Mode

SELinux supports enforcing, permissive, and disabled modes. Enforcing mode is the default. Permissive mode allows operations that are not permitted in enforcing mode and logs those operations to the SELinux audit log. Permissive mode is typically used when developing policies or troubleshooting. In disabled mode, polices are not enforced, and contexts are not applied to system objects, which makes it difficult to enable SELinux later.

To view the current SELinux mode, use the `sestatus` command mentioned previously or the `getenforce` utility.

```
$> getenforce
Enforcing
```

To change the SELinux mode, use the `setenforce` utility:

```
$> setenforce 0
$> getenforce
Permissive
```

```
$> setenforce 1
$> getenforce
Enforcing
```

Changes made with `setenforce` are lost when you restart the system. To permanently change the SELinux mode, edit the `/etc/selinux/config` file and restart the system.

6.7.3 MySQL Server SELinux Policies

MySQL Server SELinux policy modules are typically installed by default. You can view installed modules using the `semodule -l` command. MySQL Server SELinux policy modules include:

- `mysqld_selinux`
- `mysqld_safe_selinux`

For information about MySQL Server SELinux policy modules, refer to the SELinux manual pages. The manual pages provide information about types and Booleans associated with the MySQL service. Manual pages are named in the *`service-name`*`_selinux` format.

```
man mysqld_selinux
```

If SELinux manual pages are not available, refer to your distribution's SELinux documentation for information about how to generate manual pages using the `sepolicy manpage` utility.

6.7.4 SELinux File Context

The MySQL Server reads from and writes to many files. If the SELinux context is not set correctly for these files, access to the files could be denied.

The instructions that follow use the `semanage` binary to manage file context; on RHEL, it's part of the `policycoreutils-python-utils` package:

```
yum install -y policycoreutils-python-utils
```

After installing the `semanage` binary, you can list MySQL file contexts using `semanage` with the `fcontext` option.

```
semanage fcontext -l | grep -i mysql
```

Setting the MySQL Data Directory Context

The default data directory location is `/var/lib/mysql/`; and the SELinux context used is `mysqld_db_t`.

If you edit the configuration file to use a different location for the data directory, or for any of the files normally in the data directory (such as the binary logs), you may need to set the context for the new location. For example:

```
semanage fcontext -a -t mysqld_db_t "/path/to/my/custom/datadir(/.*)?"
restorecon -Rv /path/to/my/custom/datadir

semanage fcontext -a -t mysqld_db_t "/path/to/my/custom/logdir(/.*)?"
restorecon -Rv /path/to/my/custom/logdir
```

Setting the MySQL Error Log File Context

The default location for RedHat RPMs is `/var/log/mysqld.log`; and the SELinux context type used is `mysqld_log_t`.

If you edit the configuration file to use a different location, you may need to set the context for the new location. For example:

```
semanage fcontext -a -t mysqld_log_t "/path/to/my/custom/error.log"
restorecon -Rv /path/to/my/custom/error.log
```

Setting the PID File Context

The default location for the PID file is `/var/run/mysqld/mysqld.pid`; and the SELinux context type used is `mysqld_var_run_t`.

If you edit the configuration file to use a different location, you may need to set the context for the new location. For example:

```
semanage fcontext -a -t mysqld_var_run_t "/path/to/my/custom/pidfile/directory/.*?"
restorecon -Rv /path/to/my/custom/pidfile/directory
```

Setting the Unix Domain Socket Context

The default location for the Unix domain socket is `/var/lib/mysql/mysql.sock`; and the SELinux context type used is `mysqld_var_run_t`.

If you edit the configuration file to use a different location, you may need to set the context for the new location. For example:

```
semanage fcontext -a -t mysqld_var_run_t "/path/to/my/custom/mysql\.sock"
restorecon -Rv /path/to/my/custom/mysql.sock
```

Setting the secure_file_priv Directory Context

For MySQL versions since 5.6.34, 5.7.16, and 8.0.11.

Installing the MySQL Server RPM creates a `/var/lib/mysql-files/` directory but does not set the SELinux context for it. The `/var/lib/mysql-files/` directory is intended to be used for operations such as `SELECT ... INTO OUTFILE`.

If you enabled the use of this directory by setting `secure_file_priv`, you may need to set the context like so:

```
semanage fcontext -a -t mysqld_db_t "/var/lib/mysql-files/(/.*)?"
restorecon -Rv /var/lib/mysql-files
```

Edit this path if you used a different location. For security purposes, this directory should never be within the data directory.

For more information about this variable, see the `secure_file_priv` documentation.

6.7.5 SELinux TCP Port Context

The instructions that follow use the `semanage` binary to manage port context; on RHEL, it's part of the `policycoreutils-python-utils` package:

```
yum install -y policycoreutils-python-utils
```

After installing the `semanage` binary, you can list ports defined with the `mysqld_port_t` context using `semanage` with the `port` option.

```
$> semanage port -l | grep mysqld
mysqld_port_t                  tcp      1186, 3306, 63132-63164
```

6.7.5.1 Setting the TCP Port Context for mysqld

The default TCP port for `mysqld` is `3306`; and the SELinux context type used is `mysqld_port_t`.

If you configure `mysqld` to use a different TCP `port`, you may need to set the context for the new port. For example to define the SELinux context for a non-default port such as port 3307:

```
semanage port -a -t mysqld_port_t -p tcp 3307
```

To confirm that the port is added:

```
$> semanage port -l | grep mysqld
mysqld_port_t                  tcp      3307, 1186, 3306, 63132-63164
```

6.7.5.2 Setting the TCP Port Context for MySQL Features

If you enable certain MySQL features, you might need to set the SELinux TCP port context for additional ports used by those features. If ports used by MySQL features do not have the correct SELinux context, the features might not function correctly.

The following sections describe how to set port contexts for MySQL features. Generally, the same method can be used to set the port context for any MySQL features. For information about ports used by MySQL features, refer to the MySQL Port Reference.

From MySQL 8.0.14 to MySQL 8.0.17, the `mysql_connect_any` SELinux boolean must be set to `ON`. As of MySQL 8.0.18, enabling `mysql_connect_any` is not required or recommended.

```
setsebool -P mysql_connect_any=ON
```

Setting the TCP Port Context for Group Replication

If SELinux is enabled, you must set the port context for the Group Replication communication port, which is defined by the `group_replication_local_address` variable. `mysqld` must be able to bind to the Group Replication communication port and listen there. InnoDB Cluster relies on Group Replication so this applies equally to instances used in a cluster. To view ports currently used by MySQL, issue:

```
semanage port -l | grep mysqld
```

Assuming the Group Replication communication port is 33061, set the port context by issuing:

```
semanage port -a -t mysqld_port_t -p tcp 33061
```

Setting the TCP Port Context for Document Store

If SELinux is enabled, you must set the port context for the communication port used by X Plugin, which is defined by the `mysqlx_port` variable. `mysqld` must be able to bind to the X Plugin communication port and listen there.

Assuming the X Plugin communication port is 33060, set the port context by issuing:

```
semanage port -a -t mysqld_port_t -p tcp 33060
```

Setting the TCP Port Context for MySQL Router

If SELinux is enabled, you must set the port context for the communication ports used by MySQL Router. Assuming the additional communication ports used by MySQL Router are the default 6446, 6447, 64460 and 64470, on each instance set the port context by issuing:

```
semanage port -a -t mysqld_port_t -p tcp 6446
semanage port -a -t mysqld_port_t -p tcp 6447
semanage port -a -t mysqld_port_t -p tcp 64460
semanage port -a -t mysqld_port_t -p tcp 64470
```

6.7.6 Troubleshooting SELinux

Troubleshooting SELinux typically involves placing SELinux into permissive mode, rerunning problematic operations, checking for access denial messages in the SELinux audit log, and placing SELinux back into enforcing mode after problems are resolved.

To avoid placing the entire system into permissive mode using `setenforce`, you can permit only the MySQL service to run permissively by placing its SELinux domain (`mysqld_t`) into permissive mode using the `semanage` command:

```
semanage permissive -a mysqld_t
```

When you are finished troubleshooting, use this command to place the `mysqld_t` domain back into enforcing mode:

```
semanage permissive -d mysqld_t
```

SELinux writes logs for denied operations to `/var/log/audit/audit.log`. You can check for denials by searching for “denied” messages.

```
grep "denied" /var/log/audit/audit.log
```

The following sections describes a few common areas where SELinux-related issues may be encountered.

File Contexts

If a MySQL directory or file has an incorrect SELinux context, access may be denied. This issue can occur if MySQL is configured to read from or write to a non-default directory or file. For example, if you configure MySQL to use a non-default data directory, the directory may not have the expected SELinux context.

Attempting to start the MySQL service on a non-default data directory with an invalid SELinux context causes the following startup failure.

```
$> systemctl start mysql.service
```

```
Job for mysqld.service failed because the control process exited with error code.
See "systemctl status mysqld.service" and "journalctl -xe" for details.
```

In this case, a “denial” message is logged to `/var/log/audit/audit.log`:

```
$> grep "denied" /var/log/audit/audit.log
type=AVC msg=audit(1587133719.786:194): avc:  denied  { write } for  pid=7133 comm="mysqld"
name="mysql" dev="dm-0" ino=51347078 scontext=system_u:system_r:mysqld_t:s0
tcontext=unconfined_u:object_r:default_t:s0 tclass=dir permissive=0
```

For information about setting the proper SELinux context for MySQL directories and files, see Section 6.7.4, “SELinux File Context”.

Port Access

SELinux expects services such as MySQL Server to use specific ports. Changing ports without updating the SELinux policies may cause a service failure.

The `mysqld_port_t` port type defines the ports that the MySQL listens on. If you configure the MySQL Server to use a non-default port, such as port 3307, and do not update the policy to reflect the change, the MySQL service fails to start:

```
$> systemctl start mysqld.service
Job for mysqld.service failed because the control process exited with error code.
See "systemctl status mysqld.service" and "journalctl -xe" for details.
```

In this case, a denial message is logged to `/var/log/audit/audit.log`:

```
$> grep "denied" /var/log/audit/audit.log
type=AVC msg=audit(1587134375.845:198): avc:  denied  { name_bind } for  pid=7340
comm="mysqld" src=3307 scontext=system_u:system_r:mysqld_t:s0
tcontext=system_u:object_r:unreserved_port_t:s0 tclass=tcp_socket permissive=0
```

For information about setting the proper SELinux port context for MySQL, see Section 6.7.5, “SELinux TCP Port Context”. Similar port access issues can occur when enabling MySQL features that use ports that are not defined with the required context. For more information, see Section 6.7.5.2, “Setting the TCP Port Context for MySQL Features”.

Application Changes

SELinux may not be aware of application changes. For example, a new release, an application extension, or a new feature may access system resources in a way that is not permitted by SELinux, resulting in access denials. In such cases, you can use the `audit2allow` utility to create custom policies to permit access where it is required. The typical method for creating custom policies is to change the SELinux mode to permissive, identify access denial messages in the SELinux audit log, and use the `audit2allow` utility to create custom policies to permit access.

For information about using the `audit2allow` utility, refer to your distribution's SELinux documentation.

If you encounter access issues for MySQL that you believe should be handled by standard MySQL SELinux policy modules, please open a bug report in your distribution's bug tracking system.

6.8 FIPS Support

MySQL supports FIPS mode, if compiled using OpenSSL 1.0.2, and an OpenSSL library and FIPS Object Module are available at runtime.

FIPS mode on the server side applies to cryptographic operations performed by the server. This includes replication (source/replica and Group Replication) and X Plugin, which run within the server. FIPS mode also applies to attempts by clients to connect to the server.

The following sections describe FIPS mode and how to take advantage of it within MySQL:

FIPS Overview

Federal Information Processing Standards 140-2 (FIPS 140-2) describes a security standard that can be required by Federal (US Government) agencies for cryptographic modules used to protect sensitive or valuable information. To be considered acceptable for such Federal use, a cryptographic module must be certified for FIPS 140-2. If a system intended to protect sensitive data lacks the proper FIPS 140-2 certificate, Federal agencies cannot purchase it.

Products such as OpenSSL can be used in FIPS mode, although the OpenSSL library itself is not validated for FIPS. Instead, the OpenSSL library is used with the OpenSSL FIPS Object Module to enable OpenSSL-based applications to operate in FIPS mode.

For general information about FIPS and its implementation in OpenSSL, these references may be helpful:

- National Institute of Standards and Technology FIPS PUB 140-2
- OpenSSL FIPS 140-2 Security Policy
- User Guide for the OpenSSL FIPS Object Module v2.0

Important

FIPS mode imposes conditions on cryptographic operations such as restrictions on acceptable encryption algorithms or requirements for longer key lengths. For OpenSSL, the exact FIPS behavior depends on the OpenSSL version. For details, refer to the OpenSSL FIPS User Guide.

System Requirements for FIPS Mode in MySQL

For MySQL to support FIPS mode, these system requirements must be satisfied:

- At build time, MySQL must be compiled using OpenSSL. FIPS mode cannot be used in MySQL if compilation uses an SSL library different from OpenSSL.

  In addition, MySQL must be compiled with an OpenSSL version that is certified for use with FIPS. OpenSSL 1.0.2 is certified, but OpenSSL 1.1.1 is not. Binary distributions for recent versions of MySQL are compiled using OpenSSL 1.1.1 on some platforms, which means they are not certified for FIPS. This leads to tradeoffs in available MySQL features, depending on system and MySQL configuration:

  - Use a system that has OpenSSL 1.0.2 and the required FIPS Object Module. In this case, you can enable FIPS mode for MySQL if you use a binary distribution compiled using OpenSSL 1.0.2, or compile MySQL from source using OpenSSL 1.0.2. However, in this case, you cannot use the TLSv1.3 protocol or ciphersuites (which require OpenSSL 1.1.1). In addition, you are using an OpenSSL version that reached End of Life status at the end of 2019.
  - Use a system that has OpenSSL 1.1.1 or higher. In this case, you can install MySQL using binary packages, and you can use the TLSv1.3 protocol and ciphersuites, in addition to other already supported TLS protocols. However, you cannot enable FIPS mode for MySQL.

- At runtime, the OpenSSL library and OpenSSL FIPS Object Module must be available as shared (dynamically linked) objects. It is possible to build statically linked OpenSSL objects, but MySQL cannot use them.

FIPS mode has been tested for MySQL on EL7, but may work on other systems.

If your platform or operating system provides the OpenSSL FIPS Object Module, you can use it. Otherwise, you can build the OpenSSL library and FIPS Object Module from source. Use the instructions in the OpenSSL FIPS User Guide (see FIPS Overview).

Configuring FIPS Mode in MySQL

MySQL enables control of FIPS mode on the server side and the client side:

- The `ssl_fips_mode` system variable controls whether the server operates in FIPS mode.
- The `--ssl-fips-mode` client option controls whether a given MySQL client operates in FIPS mode.

The `ssl_fips_mode` system variable and `--ssl-fips-mode` client option permit these values:

- `OFF`: Disable FIPS mode.
- `ON`: Enable FIPS mode.
- `STRICT`: Enable “strict” FIPS mode.

On the server side, numeric `ssl_fips_mode` values of 0, 1, and 2 are equivalent to `OFF`, `ON`, and `STRICT`, respectively.

Important

In general, `STRICT` imposes more restrictions than `ON`, but MySQL itself has no FIPS-specific code other than to specify to OpenSSL the FIPS mode value. The exact behavior of FIPS mode for `ON` or `STRICT` depends on the OpenSSL version. For details, refer to the OpenSSL FIPS User Guide (see FIPS Overview).

Note

If the OpenSSL FIPS Object Module is not available, the only permitted value for `ssl_fips_mode` and `--ssl-fips-mode` is `OFF`. An error occurs for attempts to set the FIPS mode to a different value.

FIPS mode on the server side applies to cryptographic operations performed by the server. This includes replication (source/replica and Group Replication) and X Plugin, which run within the server.

FIPS mode also applies to attempts by clients to connect to the server. When enabled, on either the client or server side, it restricts which of the supported encryption ciphers can be chosen. However, enabling FIPS mode does not require that an encrypted connection must be used, or that user credentials must be encrypted. For example, if FIPS mode is enabled, stronger cryptographic algorithms are required. In particular, MD5 is restricted, so trying to establish an encrypted connection using an encryption cipher such as `RC4-MD5` does not work. But there is nothing about FIPS mode that prevents establishing an unencrypted connection. (To do that, you can use the `REQUIRE` clause for `CREATE USER` or `ALTER USER` for specific user accounts, or set the `require_secure_transport` system variable to affect all accounts.)

Chapter 7 Backup and Recovery

Table of Contents

It is important to back up your databases so that you can recover your data and be up and running again in case problems occur, such as system crashes, hardware failures, or users deleting data by mistake. Backups are also essential as a safeguard before upgrading a MySQL installation, and they can be used to transfer a MySQL installation to another system or to set up replica servers.

MySQL offers a variety of backup strategies from which you can choose the methods that best suit the requirements for your installation. This chapter discusses several backup and recovery topics with which you should be familiar:

- Types of backups: Logical versus physical, full versus incremental, and so forth.
- Methods for creating backups.
- Recovery methods, including point-in-time recovery.
- Backup scheduling, compression, and encryption.
- Table maintenance, to enable recovery of corrupt tables.

Additional Resources

Resources related to backup or to maintaining data availability include the following:

- Customers of MySQL Enterprise Edition can use the MySQL Enterprise Backup product for backups. For an overview of the MySQL Enterprise Backup product, see Section 30.2, “MySQL Enterprise Backup Overview”.
- A forum dedicated to backup issues is available at https://forums.mysql.com/list.php?28.
- Details for `mysqldump` can be found in Chapter 4, *MySQL Programs*.
- The syntax of the SQL statements described here is given in Chapter 13, *SQL Statements*.

- For additional information about `InnoDB` backup procedures, see Section 15.18.1, “InnoDB Backup”.
- Replication enables you to maintain identical data on multiple servers. This has several benefits, such as enabling client query load to be distributed over servers, availability of data even if a given server is taken offline or fails, and the ability to make backups with no impact on the source by using a replica. See Chapter 17, *Replication*.
- MySQL InnoDB Cluster is a collection of products that work together to provide a high availability solution. A group of MySQL servers can be configured to create a cluster using MySQL Shell. The cluster of servers has a single source, called the primary, which acts as the read-write source. Multiple secondary servers are replicas of the source. A minimum of three servers are required to create a high availability cluster. A client application is connected to the primary via MySQL Router. If the primary fails, a secondary is automatically promoted to the role of primary, and MySQL Router routes requests to the new primary.
- NDB Cluster provides a high-availability, high-redundancy version of MySQL adapted for the distributed computing environment. See Chapter 23, *MySQL NDB Cluster 8.0*, which provides information about MySQL NDB Cluster 8.0.

7.1 Backup and Recovery Types

This section describes the characteristics of different types of backups.

Physical (Raw) Versus Logical Backups

Physical backups consist of raw copies of the directories and files that store database contents. This type of backup is suitable for large, important databases that need to be recovered quickly when problems occur.

Logical backups save information represented as logical database structure (`CREATE DATABASE`, `CREATE TABLE` statements) and content (`INSERT` statements or delimited-text files). This type of backup is suitable for smaller amounts of data where you might edit the data values or table structure, or recreate the data on a different machine architecture.

Physical backup methods have these characteristics:

- The backup consists of exact copies of database directories and files. Typically this is a copy of all or part of the MySQL data directory.
- Physical backup methods are faster than logical because they involve only file copying without conversion.
- Output is more compact than for logical backup.
- Because backup speed and compactness are important for busy, important databases, the MySQL Enterprise Backup product performs physical backups. For an overview of the MySQL Enterprise Backup product, see Section 30.2, “MySQL Enterprise Backup Overview”.
- Backup and restore granularity ranges from the level of the entire data directory down to the level of individual files. This may or may not provide for table-level granularity, depending on storage engine. For example, `InnoDB` tables can each be in a separate file, or share file storage with other `InnoDB` tables; each `MyISAM` table corresponds uniquely to a set of files.
- In addition to databases, the backup can include any related files such as log or configuration files.
- Data from `MEMORY` tables is tricky to back up this way because their contents are not stored on disk. (The MySQL Enterprise Backup product has a feature where you can retrieve data from `MEMORY` tables during a backup.)
- Backups are portable only to other machines that have identical or similar hardware characteristics.

- Backups can be performed while the MySQL server is not running. If the server is running, it is necessary to perform appropriate locking so that the server does not change database contents during the backup. MySQL Enterprise Backup does this locking automatically for tables that require it.
- Physical backup tools include the `mysqlbackup` of MySQL Enterprise Backup for `InnoDB` or any other tables, or file system-level commands (such as `cp`, `scp`, `tar`, `rsync`) for `MyISAM` tables.
- For restore:
  - MySQL Enterprise Backup restores `InnoDB` and other tables that it backed up.
  - `ndb_restore` restores `NDB` tables.
  - Files copied at the file system level can be copied back to their original locations with file system commands.

Logical backup methods have these characteristics:

- The backup is done by querying the MySQL server to obtain database structure and content information.
- Backup is slower than physical methods because the server must access database information and convert it to logical format. If the output is written on the client side, the server must also send it to the backup program.
- Output is larger than for physical backup, particularly when saved in text format.
- Backup and restore granularity is available at the server level (all databases), database level (all tables in a particular database), or table level. This is true regardless of storage engine.
- The backup does not include log or configuration files, or other database-related files that are not part of databases.
- Backups stored in logical format are machine independent and highly portable.
- Logical backups are performed with the MySQL server running. The server is not taken offline.
- Logical backup tools include the `mysqldump` program and the `SELECT ... INTO OUTFILE` statement. These work for any storage engine, even `MEMORY`.
- To restore logical backups, SQL-format dump files can be processed using the `mysql` client. To load delimited-text files, use the `LOAD DATA` statement or the `mysqlimport` client.

Online Versus Offline Backups

Online backups take place while the MySQL server is running so that the database information can be obtained from the server. Offline backups take place while the server is stopped. This distinction can also be described as “hot” versus “cold” backups; a “warm” backup is one where the server remains running but locked against modifying data while you access database files externally.

Online backup methods have these characteristics:

- The backup is less intrusive to other clients, which can connect to the MySQL server during the backup and may be able to access data depending on what operations they need to perform.
- Care must be taken to impose appropriate locking so that data modifications do not take place that would compromise backup integrity. The MySQL Enterprise Backup product does such locking automatically.

Offline backup methods have these characteristics:

- Clients can be affected adversely because the server is unavailable during backup. For that reason, such backups are often taken from a replica that can be taken offline without harming availability.

- The backup procedure is simpler because there is no possibility of interference from client activity.

A similar distinction between online and offline applies for recovery operations, and similar characteristics apply. However, it is more likely for clients to be affected by online recovery than by online backup because recovery requires stronger locking. During backup, clients might be able to read data while it is being backed up. Recovery modifies data and does not just read it, so clients must be prevented from accessing data while it is being restored.

Local Versus Remote Backups

A local backup is performed on the same host where the MySQL server runs, whereas a remote backup is done from a different host. For some types of backups, the backup can be initiated from a remote host even if the output is written locally on the server. host.

- `mysqldump` can connect to local or remote servers. For SQL output (`CREATE` and `INSERT` statements), local or remote dumps can be done and generate output on the client. For delimited-text output (with the `--tab` option), data files are created on the server host.
- `SELECT ... INTO OUTFILE` can be initiated from a local or remote client host, but the output file is created on the server host.
- Physical backup methods typically are initiated locally on the MySQL server host so that the server can be taken offline, although the destination for copied files might be remote.

Snapshot Backups

Some file system implementations enable “snapshots” to be taken. These provide logical copies of the file system at a given point in time, without requiring a physical copy of the entire file system. (For example, the implementation may use copy-on-write techniques so that only parts of the file system modified after the snapshot time need be copied.) MySQL itself does not provide the capability for taking file system snapshots. It is available through third-party solutions such as Veritas, LVM, or ZFS.

Full Versus Incremental Backups

A full backup includes all data managed by a MySQL server at a given point in time. An incremental backup consists of the changes made to the data during a given time span (from one point in time to another). MySQL has different ways to perform full backups, such as those described earlier in this section. Incremental backups are made possible by enabling the server's binary log, which the server uses to record data changes.

Full Versus Point-in-Time (Incremental) Recovery

A full recovery restores all data from a full backup. This restores the server instance to the state that it had when the backup was made. If that state is not sufficiently current, a full recovery can be followed by recovery of incremental backups made since the full backup, to bring the server to a more up-to-date state.

Incremental recovery is recovery of changes made during a given time span. This is also called point-in-time recovery because it makes a server's state current up to a given time. Point-in-time recovery is based on the binary log and typically follows a full recovery from the backup files that restores the server to its state when the backup was made. Then the data changes written in the binary log files are applied as incremental recovery to redo data modifications and bring the server up to the desired point in time.

Table Maintenance

Data integrity can be compromised if tables become corrupt. For `InnoDB` tables, this is not a typical issue. For programs to check `MyISAM` tables and repair them if problems are found, see Section 7.6, “MyISAM Table Maintenance and Crash Recovery”.

Backup Scheduling, Compression, and Encryption

Backup scheduling is valuable for automating backup procedures. Compression of backup output reduces space requirements, and encryption of the output provides better security against unauthorized access of backed-up data. MySQL itself does not provide these capabilities. The MySQL Enterprise Backup product can compress `InnoDB` backups, and compression or encryption of backup output can be achieved using file system utilities. Other third-party solutions may be available.

7.2 Database Backup Methods

This section summarizes some general methods for making backups.

Making a Hot Backup with MySQL Enterprise Backup

Customers of MySQL Enterprise Edition can use the MySQL Enterprise Backup product to do physical backups of entire instances or selected databases, tables, or both. This product includes features for incremental and compressed backups. Backing up the physical database files makes restore much faster than logical techniques such as the `mysqldump` command. `InnoDB` tables are copied using a hot backup mechanism. (Ideally, the `InnoDB` tables should represent a substantial majority of the data.) Tables from other storage engines are copied using a warm backup mechanism. For an overview of the MySQL Enterprise Backup product, see Section 30.2, “MySQL Enterprise Backup Overview”.

Making Backups with mysqldump

The `mysqldump` program can make backups. It can back up all kinds of tables. (See Section 7.4, “Using mysqldump for Backups”.)

For `InnoDB` tables, it is possible to perform an online backup that takes no locks on tables using the `--single-transaction` option to `mysqldump`. See Section 7.3.1, “Establishing a Backup Policy”.

Making Backups by Copying Table Files

MyISAM tables can be backed up by copying table files (`*.MYD`, `*.MYI` files, and associated `*.sdi` files). To get a consistent backup, stop the server or lock and flush the relevant tables:

```
FLUSH TABLES tbl_list WITH READ LOCK;
```

You need only a read lock; this enables other clients to continue to query the tables while you are making a copy of the files in the database directory. The flush is needed to ensure that the all active index pages are written to disk before you start the backup. See Section 13.3.6, “LOCK TABLES and UNLOCK TABLES Statements”, and Section 13.7.8.3, “FLUSH Statement”.

You can also create a binary backup simply by copying the table files, as long as the server isn't updating anything. (But note that table file copying methods do not work if your database contains `InnoDB` tables. Also, even if the server is not actively updating data, `InnoDB` may still have modified data cached in memory and not flushed to disk.)

For an example of this backup method, refer to the export and import example in Section 13.2.6, “IMPORT TABLE Statement”.

Making Delimited-Text File Backups

To create a text file containing a table's data, you can use `SELECT * INTO OUTFILE 'file_name' FROM tbl_name`. The file is created on the MySQL server host, not the client host. For this statement, the output file cannot already exist because permitting files to be overwritten constitutes a security risk. See Section 13.2.13, “SELECT Statement”. This method works for any kind of data file, but saves only table data, not the table structure.

Another way to create text data files (along with files containing `CREATE TABLE` statements for the backed up tables) is to use `mysqldump` with the `--tab` option. See Section 7.4.3, “Dumping Data in Delimited-Text Format with mysqldump”.

To reload a delimited-text data file, use `LOAD DATA` or `mysqlimport`.

Making Incremental Backups by Enabling the Binary Log

MySQL supports incremental backups using the binary log. The binary log files provide you with the information you need to replicate changes to the database that are made subsequent to the point at which you performed a backup. Therefore, to allow a server to be restored to a point-in-time, binary logging must be enabled on it, which is the default setting for MySQL 8.0 ; see Section 5.4.4, “The Binary Log”.

At the moment you want to make an incremental backup (containing all changes that happened since the last full or incremental backup), you should rotate the binary log by using `FLUSH LOGS`. This done, you need to copy to the backup location all binary logs which range from the one of the moment of the last full or incremental backup to the last but one. These binary logs are the incremental backup; at restore time, you apply them as explained in Section 7.5, “Point-in-Time (Incremental) Recovery”. The next time you do a full backup, you should also rotate the binary log using `FLUSH LOGS` or `mysqldump --flush-logs`. See Section 4.5.4, “mysqldump — A Database Backup Program”.

Making Backups Using Replicas

If you have performance problems with a server while making backups, one strategy that can help is to set up replication and perform backups on the replica rather than on the source. See Section 17.4.1, “Using Replication for Backups”.

If you are backing up a replica, you should back up its connection metadata repository and applier metadata repository (see Section 17.2.4, “Relay Log and Replication Metadata Repositories”) when you back up the replica's databases, regardless of the backup method you choose. This information is always needed to resume replication after you restore the replica's data. If your replica is replicating `LOAD DATA` statements, you should also back up any `SQL_LOAD-*` files that exist in the directory that the replica uses for this purpose. The replica needs these files to resume replication of any interrupted `LOAD DATA` operations. The location of this directory is the value of the system variable `replica_load_tmpdir` (from MySQL 8.0.26) or `slave_load_tmpdir` (before MySQL 8.0.26). If the server was not started with that variable set, the directory location is the value of the `tmpdir` system variable.

Recovering Corrupt Tables

If you have to restore `MyISAM` tables that have become corrupt, try to recover them using `REPAIR TABLE` or `myisamchk -r` first. That should work in 99.9% of all cases. If `myisamchk` fails, see Section 7.6, “MyISAM Table Maintenance and Crash Recovery”.

Making Backups Using a File System Snapshot

If you are using a Veritas file system, you can make a backup like this:

1. From a client program, execute `FLUSH TABLES WITH READ LOCK`.
2. From another shell, execute `mount vxfs snapshot`.
3. From the first client, execute `UNLOCK TABLES`.
4. Copy files from the snapshot.
5. Unmount the snapshot.

Similar snapshot capabilities may be available in other file systems, such as LVM or ZFS.

7.3 Example Backup and Recovery Strategy

This section discusses a procedure for performing backups that enables you to recover data after several types of crashes:

- Operating system crash
- Power failure
- File system crash
- Hardware problem (hard drive, motherboard, and so forth)

The example commands do not include options such as `--user` and `--password` for the `mysqldump` and `mysql` client programs. You should include such options as necessary to enable client programs to connect to the MySQL server.

Assume that data is stored in the `InnoDB` storage engine, which has support for transactions and automatic crash recovery. Assume also that the MySQL server is under load at the time of the crash. If it were not, no recovery would ever be needed.

For cases of operating system crashes or power failures, we can assume that MySQL's disk data is available after a restart. The `InnoDB` data files might not contain consistent data due to the crash, but `InnoDB` reads its logs and finds in them the list of pending committed and noncommitted transactions that have not been flushed to the data files. `InnoDB` automatically rolls back those transactions that were not committed, and flushes to its data files those that were committed. Information about this recovery process is conveyed to the user through the MySQL error log. The following is an example log excerpt:

```
InnoDB: Database was not shut down normally.
InnoDB: Starting recovery from log files...
InnoDB: Starting log scan based on checkpoint at
InnoDB: log sequence number 0 13674004
InnoDB: Doing recovery: scanned up to log sequence number 0 13739520
InnoDB: Doing recovery: scanned up to log sequence number 0 13805056
InnoDB: Doing recovery: scanned up to log sequence number 0 13870592
InnoDB: Doing recovery: scanned up to log sequence number 0 13936128
...
InnoDB: Doing recovery: scanned up to log sequence number 0 20555264
InnoDB: Doing recovery: scanned up to log sequence number 0 20620800
InnoDB: Doing recovery: scanned up to log sequence number 0 20664692
InnoDB: 1 uncommitted transaction(s) which must be rolled back
InnoDB: Starting rollback of uncommitted transactions
InnoDB: Rolling back trx no 16745
InnoDB: Rolling back of trx no 16745 completed
InnoDB: Rollback of uncommitted transactions completed
InnoDB: Starting an apply batch of log records to the database...
InnoDB: Apply batch completed
InnoDB: Started
mysqld: ready for connections
```

For the cases of file system crashes or hardware problems, we can assume that the MySQL disk data is *not* available after a restart. This means that MySQL fails to start successfully because some blocks of disk data are no longer readable. In this case, it is necessary to reformat the disk, install a new one, or otherwise correct the underlying problem. Then it is necessary to recover our MySQL data from backups, which means that backups must already have been made. To make sure that is the case, design and implement a backup policy.

7.3.1 Establishing a Backup Policy

To be useful, backups must be scheduled regularly. A full backup (a snapshot of the data at a point in time) can be done in MySQL with several tools. For example, MySQL Enterprise Backup can perform a physical backup of an entire instance, with optimizations to minimize overhead and avoid disruption

when backing up InnoDB data files; mysqldump provides online logical backup. This discussion uses mysqldump.

Assume that we make a full backup of all our InnoDB tables in all databases using the following command on Sunday at 1 p.m., when load is low:

```
$> mysqldump --all-databases --master-data --single-transaction > backup_sunday_1_PM.sql
```

The resulting .sql file produced by mysqldump contains a set of SQL INSERT statements that can be used to reload the dumped tables at a later time.

This backup operation acquires a global read lock on all tables at the beginning of the dump (using FLUSH TABLES WITH READ LOCK). As soon as this lock has been acquired, the binary log coordinates are read and the lock is released. If long updating statements are running when the FLUSH statement is issued, the backup operation may stall until those statements finish. After that, the dump becomes lock-free and does not disturb reads and writes on the tables.

It was assumed earlier that the tables to back up are InnoDB tables, so --single-transaction uses a consistent read and guarantees that data seen by mysqldump does not change. (Changes made by other clients to InnoDB tables are not seen by the mysqldump process.) If the backup operation includes nontransactional tables, consistency requires that they do not change during the backup. For example, for the MyISAM tables in the mysql database, there must be no administrative changes to MySQL accounts during the backup.

Full backups are necessary, but it is not always convenient to create them. They produce large backup files and take time to generate. They are not optimal in the sense that each successive full backup includes all data, even that part that has not changed since the previous full backup. It is more efficient to make an initial full backup, and then to make incremental backups. The incremental backups are smaller and take less time to produce. The tradeoff is that, at recovery time, you cannot restore your data just by reloading the full backup. You must also process the incremental backups to recover the incremental changes.

To make incremental backups, we need to save the incremental changes. In MySQL, these changes are represented in the binary log, so the MySQL server should always be started with the --log-bin option to enable that log. With binary logging enabled, the server writes each data change into a file while it updates data. Looking at the data directory of a MySQL server that has been running for some days, we find these MySQL binary log files:

```
-rw-rw---- 1 guilhem  guilhem   1277324 Nov 10 23:59 gbichot2-bin.000001
-rw-rw---- 1 guilhem  guilhem         4 Nov 10 23:59 gbichot2-bin.000002
-rw-rw---- 1 guilhem  guilhem        79 Nov 11 11:06 gbichot2-bin.000003
-rw-rw---- 1 guilhem  guilhem       508 Nov 11 11:08 gbichot2-bin.000004
-rw-rw---- 1 guilhem  guilhem 220047446 Nov 12 16:47 gbichot2-bin.000005
-rw-rw---- 1 guilhem  guilhem    998412 Nov 14 10:08 gbichot2-bin.000006
-rw-rw---- 1 guilhem  guilhem       361 Nov 14 10:07 gbichot2-bin.index
```

Each time it restarts, the MySQL server creates a new binary log file using the next number in the sequence. While the server is running, you can also tell it to close the current binary log file and begin a new one manually by issuing a FLUSH LOGS SQL statement or with a mysqladmin flush-logs command. mysqldump also has an option to flush the logs. The .index file in the data directory contains the list of all MySQL binary logs in the directory.

The MySQL binary logs are important for recovery because they form the set of incremental backups. If you make sure to flush the logs when you make your full backup, the binary log files created afterward contain all the data changes made since the backup. Let's modify the previous mysqldump command a bit so that it flushes the MySQL binary logs at the moment of the full backup, and so that the dump file contains the name of the new current binary log:

```
$> mysqldump --single-transaction --flush-logs --master-data=2 \
        --all-databases > backup_sunday_1_PM.sql
```

After executing this command, the data directory contains a new binary log file, gbichot2-bin.000007, because the --flush-logs option causes the server to flush its logs. The --master-

`data` option causes `mysqldump` to write binary log information to its output, so the resulting `.sql` dump file includes these lines:

```
-- Position to start replication or point-in-time recovery from
-- CHANGE MASTER TO MASTER_LOG_FILE='gbichot2-bin.000007',MASTER_LOG_POS=4;
```

Because the `mysqldump` command made a full backup, those lines mean two things:

- The dump file contains all changes made before any changes written to the `gbichot2-bin.000007` binary log file or higher.
- All data changes logged after the backup are not present in the dump file, but are present in the `gbichot2-bin.000007` binary log file or higher.

On Monday at 1 p.m., we can create an incremental backup by flushing the logs to begin a new binary log file. For example, executing a `mysqladmin flush-logs` command creates `gbichot2-bin.000008`. All changes between the Sunday 1 p.m. full backup and Monday 1 p.m. are written in `gbichot2-bin.000007`. This incremental backup is important, so it is a good idea to copy it to a safe place. (For example, back it up on tape or DVD, or copy it to another machine.) On Tuesday at 1 p.m., execute another `mysqladmin flush-logs` command. All changes between Monday 1 p.m. and Tuesday 1 p.m. are written in `gbichot2-bin.000008` (which also should be copied somewhere safe).

The MySQL binary logs take up disk space. To free up space, purge them from time to time. One way to do this is by deleting the binary logs that are no longer needed, such as when we make a full backup:

```
$> mysqldump --single-transaction --flush-logs --master-data=2 \
        --all-databases --delete-master-logs > backup_sunday_1_PM.sql
```

Note

Deleting the MySQL binary logs with `mysqldump --delete-master-logs` can be dangerous if your server is a replication source server, because replicas might not yet fully have processed the contents of the binary log. The description for the `PURGE BINARY LOGS` statement explains what should be verified before deleting the MySQL binary logs. See Section 13.4.1.1, “PURGE BINARY LOGS Statement”.

7.3.2 Using Backups for Recovery

Now, suppose that we have a catastrophic unexpected exit on Wednesday at 8 a.m. that requires recovery from backups. To recover, first we restore the last full backup we have (the one from Sunday 1 p.m.). The full backup file is just a set of SQL statements, so restoring it is very easy:

```
$> mysql < backup_sunday_1_PM.sql
```

At this point, the data is restored to its state as of Sunday 1 p.m.. To restore the changes made since then, we must use the incremental backups; that is, the `gbichot2-bin.000007` and `gbichot2-bin.000008` binary log files. Fetch the files if necessary from where they were backed up, and then process their contents like this:

```
$> mysqlbinlog gbichot2-bin.000007 gbichot2-bin.000008 | mysql
```

We now have recovered the data to its state as of Tuesday 1 p.m., but still are missing the changes from that date to the date of the crash. To not lose them, we would have needed to have the MySQL server store its MySQL binary logs into a safe location (RAID disks, SAN, ...) different from the place where it stores its data files, so that these logs were not on the destroyed disk. (That is, we can start the server with a `--log-bin` option that specifies a location on a different physical device from the one on which the data directory resides. That way, the logs are safe even if the device containing the directory is lost.) If we had done this, we would have the `gbichot2-bin.000009` file (and any

subsequent files) at hand, and we could apply them using `mysqlbinlog` and `mysql` to restore the most recent data changes with no loss up to the moment of the crash:

```
$> mysqlbinlog gbichot2-bin.000009 ... | mysql
```

For more information about using `mysqlbinlog` to process binary log files, see Section 7.5, “Point-in-Time (Incremental) Recovery”.

7.3.3 Backup Strategy Summary

In case of an operating system crash or power failure, `InnoDB` itself does all the job of recovering data. But to make sure that you can sleep well, observe the following guidelines:

- Always tun the MySQL server with binary logging enabled (that is the default setting for MySQL 8.0). If you have such safe media, this technique can also be good for disk load balancing (which results in a performance improvement).
- Make periodic full backups, using the `mysqldump` command shown earlier in Section 7.3.1, “Establishing a Backup Policy”, that makes an online, nonblocking backup.
- Make periodic incremental backups by flushing the logs with `FLUSH LOGS` or `mysqladmin flush-logs`.

7.4 Using mysqldump for Backups

Tip

Consider using the MySQL Shell dump utilities, which provide parallel dumping with multiple threads, file compression, and progress information display, as well as cloud features such as Oracle Cloud Infrastructure Object Storage streaming, and MySQL Database Service compatibility checks and modifications. Dumps can be easily imported into a MySQL Server instance or a MySQL Database Service DB System using the MySQL Shell load dump utilities. Installation instructions for MySQL Shell can be found here.

This section describes how to use `mysqldump` to produce dump files, and how to reload dump files. A dump file can be used in several ways:

- As a backup to enable data recovery in case of data loss.
- As a source of data for setting up replicas.
- As a source of data for experimentation:
  - To make a copy of a database that you can use without changing the original data.
  - To test potential upgrade incompatibilities.

`mysqldump` produces two types of output, depending on whether the `--tab` option is given:

- Without `--tab`, `mysqldump` writes SQL statements to the standard output. This output consists of `CREATE` statements to create dumped objects (databases, tables, stored routines, and so forth), and `INSERT` statements to load data into tables. The output can be saved in a file and reloaded later using `mysql` to recreate the dumped objects. Options are available to modify the format of the SQL statements, and to control which objects are dumped.
- With `--tab`, `mysqldump` produces two output files for each dumped table. The server writes one file as tab-delimited text, one line per table row. This file is named *`tbl_name`*`.txt` in the output directory. The server also sends a `CREATE TABLE` statement for the table to `mysqldump`, which writes it as a file named *`tbl_name`*`.sql` in the output directory.

7.4.1 Dumping Data in SQL Format with mysqldump

This section describes how to use `mysqldump` to create SQL-format dump files. For information about reloading such dump files, see Section 7.4.2, “Reloading SQL-Format Backups”.

By default, `mysqldump` writes information as SQL statements to the standard output. You can save the output in a file:

```
$> mysqldump [arguments] > file_name
```

To dump all databases, invoke `mysqldump` with the `--all-databases` option:

```
$> mysqldump --all-databases > dump.sql
```

To dump only specific databases, name them on the command line and use the `--databases` option:

```
$> mysqldump --databases db1 db2 db3 > dump.sql
```

The `--databases` option causes all names on the command line to be treated as database names. Without this option, `mysqldump` treats the first name as a database name and those following as table names.

With `--all-databases` or `--databases`, `mysqldump` writes `CREATE DATABASE` and `USE` statements prior to the dump output for each database. This ensures that when the dump file is reloaded, it creates each database if it does not exist and makes it the default database so database contents are loaded into the same database from which they came. If you want to cause the dump file to force a drop of each database before recreating it, use the `--add-drop-database` option as well. In this case, `mysqldump` writes a `DROP DATABASE` statement preceding each `CREATE DATABASE` statement.

To dump a single database, name it on the command line:

```
$> mysqldump --databases test > dump.sql
```

In the single-database case, it is permissible to omit the `--databases` option:

```
$> mysqldump test > dump.sql
```

The difference between the two preceding commands is that without `--databases`, the dump output contains no `CREATE DATABASE` or `USE` statements. This has several implications:

- When you reload the dump file, you must specify a default database name so that the server knows which database to reload.
- For reloading, you can specify a database name different from the original name, which enables you to reload the data into a different database.
- If the database to be reloaded does not exist, you must create it first.
- Because the output contains no `CREATE DATABASE` statement, the `--add-drop-database` option has no effect. If you use it, it produces no `DROP DATABASE` statement.

To dump only specific tables from a database, name them on the command line following the database name:

```
$> mysqldump test t1 t3 t7 > dump.sql
```

By default, if GTIDs are in use on the server where you create the dump file (`gtid_mode=ON`), `mysqldump` includes a `SET @@GLOBAL.gtid_purged` statement in the output to add the GTIDs from the `gtid_executed` set on the source server to the `gtid_purged` set on the target server. If you are dumping only specific databases or tables, it is important to note that the value that is included by `mysqldump` includes the GTIDs of all transactions in the `gtid_executed` set on the source server, even those that changed suppressed parts of the database, or other databases on the server that

were not included in the partial dump. If you only replay one partial dump file on the target server, the extra GTIDs do not cause any problems with the future operation of that server. However, if you replay a second dump file on the target server that contains the same GTIDs (for example, another partial dump from the same source server), any `SET @@GLOBAL.gtid_purged` statement in the second dump file fails. To avoid this issue, either set the `mysqldump` option `--set-gtid-purged` to `OFF` or `COMMENTED` to output the second dump file without an active `SET @@GLOBAL.gtid_purged` statement, or remove the statement manually before replaying the dump file.

7.4.2 Reloading SQL-Format Backups

To reload a dump file written by `mysqldump` that consists of SQL statements, use it as input to the `mysql` client. If the dump file was created by `mysqldump` with the `--all-databases` or `--databases` option, it contains `CREATE DATABASE` and `USE` statements and it is not necessary to specify a default database into which to load the data:

```
$> mysql < dump.sql
```

Alternatively, from within `mysql`, use a `source` command:

```
mysql> source dump.sql
```

If the file is a single-database dump not containing `CREATE DATABASE` and `USE` statements, create the database first (if necessary):

```
$> mysqladmin create db1
```

Then specify the database name when you load the dump file:

```
$> mysql db1 < dump.sql
```

Alternatively, from within `mysql`, create the database, select it as the default database, and load the dump file:

```
mysql> CREATE DATABASE IF NOT EXISTS db1;
mysql> USE db1;
mysql> source dump.sql
```

Note

For Windows PowerShell users: Because the "<" character is reserved for future use in PowerShell, an alternative approach is required, such as using quotes `cmd.exe /c "mysql < dump.sql"`.

7.4.3 Dumping Data in Delimited-Text Format with mysqldump

This section describes how to use `mysqldump` to create delimited-text dump files. For information about reloading such dump files, see Section 7.4.4, “Reloading Delimited-Text Format Backups”.

If you invoke `mysqldump` with the `--tab=dir_name` option, it uses *`dir_name`* as the output directory and dumps tables individually in that directory using two files for each table. The table name is the base name for these files. For a table named `t1`, the files are named `t1.sql` and `t1.txt`. The `.sql` file contains a `CREATE TABLE` statement for the table. The `.txt` file contains the table data, one line per table row.

The following command dumps the contents of the `db1` database to files in the `/tmp` database:

```
$> mysqldump --tab=/tmp db1
```

The `.txt` files containing table data are written by the server, so they are owned by the system account used for running the server. The server uses `SELECT ... INTO OUTFILE` to write the files, so you must have the `FILE` privilege to perform this operation, and an error occurs if a given `.txt` file already exists.

The server sends the `CREATE` definitions for dumped tables to `mysqldump`, which writes them to `.sql` files. These files therefore are owned by the user who executes `mysqldump`.

It is best that `--tab` be used only for dumping a local server. If you use it with a remote server, the `--tab` directory must exist on both the local and remote hosts, and the `.txt` files are written by the server in the remote directory (on the server host), whereas the `.sql` files are written by `mysqldump` in the local directory (on the client host).

For `mysqldump --tab`, the server by default writes table data to `.txt` files one line per row with tabs between column values, no quotation marks around column values, and newline as the line terminator. (These are the same defaults as for `SELECT ... INTO OUTFILE`.)

To enable data files to be written using a different format, `mysqldump` supports these options:

- `--fields-terminated-by=`*`str`*

  The string for separating column values (default: tab).

- `--fields-enclosed-by=`*`char`*

  The character within which to enclose column values (default: no character).

- `--fields-optionally-enclosed-by=`*`char`*

  The character within which to enclose non-numeric column values (default: no character).

- `--fields-escaped-by=`*`char`*

  The character for escaping special characters (default: no escaping).

- `--lines-terminated-by=`*`str`*

  The line-termination string (default: newline).

Depending on the value you specify for any of these options, it might be necessary on the command line to quote or escape the value appropriately for your command interpreter. Alternatively, specify the value using hex notation. Suppose that you want `mysqldump` to quote column values within double quotation marks. To do so, specify double quote as the value for the `--fields-enclosed-by` option. But this character is often special to command interpreters and must be treated specially. For example, on Unix, you can quote the double quote like this:

```
--fields-enclosed-by='"'
```

On any platform, you can specify the value in hex:

```
--fields-enclosed-by=0x22
```

It is common to use several of the data-formatting options together. For example, to dump tables in comma-separated values format with lines terminated by carriage-return/newline pairs (`\r\n`), use this command (enter it on a single line):

```
$> mysqldump --tab=/tmp --fields-terminated-by=,
        --fields-enclosed-by='"' --lines-terminated-by=0x0d0a db1
```

Should you use any of the data-formatting options to dump table data, you need to specify the same format when you reload data files later, to ensure proper interpretation of the file contents.

7.4.4 Reloading Delimited-Text Format Backups

For backups produced with `mysqldump --tab`, each table is represented in the output directory by an `.sql` file containing the `CREATE TABLE` statement for the table, and a `.txt` file containing the table data. To reload a table, first change location into the output directory. Then process the `.sql` file with `mysql` to create an empty table and process the `.txt` file to load the data into the table:

```
$> mysql db1 < t1.sql
$> mysqlimport db1 t1.txt
```

An alternative to using `mysqlimport` to load the data file is to use the `LOAD DATA` statement from within the `mysql` client:

```
mysql> USE db1;
mysql> LOAD DATA INFILE 't1.txt' INTO TABLE t1;
```

If you used any data-formatting options with `mysqldump` when you initially dumped the table, you must use the same options with `mysqlimport` or `LOAD DATA` to ensure proper interpretation of the data file contents:

```
$> mysqlimport --fields-terminated-by=,
         --fields-enclosed-by='"' --lines-terminated-by=0x0d0a db1 t1.txt
```

Or:

```
mysql> USE db1;
mysql> LOAD DATA INFILE 't1.txt' INTO TABLE t1
       FIELDS TERMINATED BY ',' FIELDS ENCLOSED BY '"'
       LINES TERMINATED BY '\r\n';
```

7.4.5 mysqldump Tips

This section surveys techniques that enable you to use `mysqldump` to solve specific problems:

- How to make a copy a database
- How to copy a database from one server to another
- How to dump stored programs (stored procedures and functions, triggers, and events)
- How to dump definitions and data separately

7.4.5.1 Making a Copy of a Database

```
$> mysqldump db1 > dump.sql
$> mysqladmin create db2
$> mysql db2 < dump.sql
```

Do not use `--databases` on the `mysqldump` command line because that causes `USE db1` to be included in the dump file, which overrides the effect of naming `db2` on the `mysql` command line.

7.4.5.2 Copy a Database from one Server to Another

On Server 1:

```
$> mysqldump --databases db1 > dump.sql
```

Copy the dump file from Server 1 to Server 2.

On Server 2:

```
$> mysql < dump.sql
```

Use of `--databases` with the `mysqldump` command line causes the dump file to include `CREATE DATABASE` and `USE` statements that create the database if it does exist and make it the default database for the reloaded data.

Alternatively, you can omit `--databases` from the `mysqldump` command. Then you need to create the database on Server 2 (if necessary) and specify it as the default database when you reload the dump file.

On Server 1:

```
$> mysqldump db1 > dump.sql
```

On Server 2:

```
$> mysqladmin create db1
$> mysql db1 < dump.sql
```

You can specify a different database name in this case, so omitting `--databases` from the `mysqldump` command enables you to dump data from one database and load it into another.

7.4.5.3 Dumping Stored Programs

Several options control how `mysqldump` handles stored programs (stored procedures and functions, triggers, and events):

- `--events`: Dump Event Scheduler events
- `--routines`: Dump stored procedures and functions
- `--triggers`: Dump triggers for tables

The `--triggers` option is enabled by default so that when tables are dumped, they are accompanied by any triggers they have. The other options are disabled by default and must be specified explicitly to dump the corresponding objects. To disable any of these options explicitly, use its skip form: `--skip-events`, `--skip-routines`, or `--skip-triggers`.

7.4.5.4 Dumping Table Definitions and Content Separately

The `--no-data` option tells `mysqldump` not to dump table data, resulting in the dump file containing only statements to create the tables. Conversely, the `--no-create-info` option tells `mysqldump` to suppress `CREATE` statements from the output, so that the dump file contains only table data.

For example, to dump table definitions and data separately for the `test` database, use these commands:

```
$> mysqldump --no-data test > dump-defs.sql
$> mysqldump --no-create-info test > dump-data.sql
```

For a definition-only dump, add the `--routines` and `--events` options to also include stored routine and event definitions:

```
$> mysqldump --no-data --routines --events test > dump-defs.sql
```

7.4.5.5 Using mysqldump to Test for Upgrade Incompatibilities

When contemplating a MySQL upgrade, it is prudent to install the newer version separately from your current production version. Then you can dump the database and database object definitions from the production server and load them into the new server to verify that they are handled properly. (This is also useful for testing downgrades.)

On the production server:

```
$> mysqldump --all-databases --no-data --routines --events > dump-defs.sql
```

On the upgraded server:

```
$> mysql < dump-defs.sql
```

Because the dump file does not contain table data, it can be processed quickly. This enables you to spot potential incompatibilities without waiting for lengthy data-loading operations. Look for warnings or errors while the dump file is being processed.

After you have verified that the definitions are handled properly, dump the data and try to load it into the upgraded server.

On the production server:

```
$> mysqldump --all-databases --no-create-info > dump-data.sql
```

On the upgraded server:

```
$> mysql < dump-data.sql
```

Now check the table contents and run some test queries.

7.5 Point-in-Time (Incremental) Recovery

Point-in-time recovery refers to recovery of data changes up to a given point in time. Typically, this type of recovery is performed after restoring a full backup that brings the server to its state as of the time the backup was made. (The full backup can be made in several ways, such as those listed in Section 7.2, “Database Backup Methods”.) Point-in-time recovery then brings the server up to date incrementally from the time of the full backup to a more recent time.

7.5.1 Point-in-Time Recovery Using Binary Log

This section explains the general idea of using the binary log to perform a point-in-time-recovery. The next section, Section 7.5.2, “Point-in-Time Recovery Using Event Positions”, explains the operation in details with an example.

Note

Many of the examples in this and the next section use the `mysql` client to process binary log output produced by `mysqlbinlog`. If your binary log contains `\0` (null) characters, that output cannot be parsed by `mysql` unless you invoke it with the `--binary-mode` option.

The source of information for point-in-time recovery is the set of binary log files generated subsequent to the full backup operation. Therefore, to allow a server to be restored to a point-in-time, binary logging must be enabled on it, which is the default setting for MySQL 8.0 (see Section 5.4.4, “The Binary Log”).

To restore data from the binary log, you must know the name and location of the current binary log files. By default, the server creates binary log files in the data directory, but a path name can be specified with the `--log-bin` option to place the files in a different location. To see a listing of all binary log files, use this statement:

```
mysql> SHOW BINARY LOGS;
```

To determine the name of the current binary log file, issue the following statement:

```
mysql> SHOW MASTER STATUS;
```

The `mysqlbinlog` utility converts the events in the binary log files from binary format to text so that they can be viewed or applied. `mysqlbinlog` has options for selecting sections of the binary log based on event times or position of events within the log. See Section 4.6.9, “mysqlbinlog — Utility for Processing Binary Log Files”.

Applying events from the binary log causes the data modifications they represent to be reexecuted. This enables recovery of data changes for a given span of time. To apply events from the binary log, process `mysqlbinlog` output using the `mysql` client:

```
$> mysqlbinlog binlog_files | mysql -u root -p
```

If binary log files have been encrypted, which can be done from MySQL 8.0.14 onwards, `mysqlbinlog` cannot read them directly as in the above example, but can read them from the server using the `--read-from-remote-server` (`-R`) option. For example:

```
$> mysqlbinlog --read-from-remote-server --host=host_name --port=3306  --user=root --password --ssl-mod
```

Here, the option `--ssl-mode=required` has been used to ensure that the data from the binary log files is protected in transit, because it is sent to `mysqlbinlog` in an unencrypted format.

Important

`VERIFY_CA` and `VERIFY_IDENTITY` are better choices than `REQUIRED` for the SSL mode, because they help prevent man-in-the-middle attacks. To implement one of these settings, you must first ensure that the CA certificate for the server is reliably available to all the clients that use it in your environment, otherwise availability issues will result. See Command Options for Encrypted Connections.

Viewing log contents can be useful when you need to determine event times or positions to select partial log contents prior to executing events. To view events from the log, send `mysqlbinlog` output into a paging program:

```
$> mysqlbinlog binlog_files | more
```

Alternatively, save the output in a file and view the file in a text editor:

```
$> mysqlbinlog binlog_files > tmpfile
$> ... edit tmpfile ...
```

After editing the file, apply the contents as follows:

```
$> mysql -u root -p < tmpfile
```

If you have more than one binary log to apply on the MySQL server, use a single connection to apply the contents of all binary log files that you want to process. Here is one way to do so:

```
$> mysqlbinlog binlog.000001 binlog.000002 | mysql -u root -p
```

Another approach is to write the whole log to a single file and then process the file:

```
$> mysqlbinlog binlog.000001 >  /tmp/statements.sql
$> mysqlbinlog binlog.000002 >> /tmp/statements.sql
$> mysql -u root -p -e "source /tmp/statements.sql"
```

7.5.2 Point-in-Time Recovery Using Event Positions

The last section, Section 7.5.1, “Point-in-Time Recovery Using Binary Log”, explains the general idea of using the binary log to perform a point-in-time-recovery. The section explains the operation in details with an example.

As an example, suppose that around 20:06:00 on March 11, 2020, an SQL statement was executed that deleted a table. You can perform a point-in-time recovery to restore the server up to its state right before the table deletion. These are some sample steps to achieve that:

1. Restore the last full backup created before the point-in-time of interest (call it t_p, which is 20:06:00 on March 11, 2020 in our example). When finished, note the binary log position up to which you have restored the server for later use, and restart the server.

Note

While the last binary log position recovered is also displayed by InnoDB after the restore and server restart, that is *not* a reliable means for obtaining the ending log position of your restore, as there could be DDL events and non-InnoDB changes that have taken place after the time reflected by the displayed position. Your backup and restore tool should provide you with the last binary log position for your recovery: for example, if you are using `mysqlbinlog` for the task, check the stop position of the binary log replay;

if you are using MySQL Enterprise Backup, the last binary log position has been saved in your backup. See Point-in-Time Recovery.

2. Find the precise binary log event position corresponding to the point in time up to which you want to restore your database. In our example, given that we know the rough time where the table deletion took place (t_p), we can find the log position by checking the log contents around that time using the `mysqlbinlog` utility. Use the `--start-datetime` and `--stop-datetime` options to specify a short time period around t_p, and then look for the event in the output. For example:

```
$> mysqlbinlog --start-datetime="2020-03-11 20:05:00" \
                   --stop-datetime="2020-03-11 20:08:00" --verbose \
         /var/lib/mysql/bin.123456 | grep -C 15 "DROP TABLE"

/*!80014 SET @@session.original_server_version=80019*//*!*/;
/*!80014 SET @@session.immediate_server_version=80019*//*!*/;
SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
# at 232
#200311 20:06:20 server id 1  end_log_pos 355 CRC32 0x2fc1e5ea  Query thread_id=16 exec_time=0 error_co
SET TIMESTAMP=1583971580/*!*/;
SET @@session.pseudo_thread_id=16/*!*/;
SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0, @@session.unique_checks=1, @@session.
SET @@session.sql_mode=1168113696/*!*/;
SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
/*!\C utf8mb4 *//*!*/;
SET @@session.character_set_client=255,@@session.collation_connection=255,@@session.collation_server=25
SET @@session.lc_time_names=0/*!*/;
SET @@session.collation_database=DEFAULT/*!*/;
/*!80011 SET @@session.default_collation_for_utf8mb4=255*//*!*/;
DROP TABLE `pets`.`cats` /* generated by server */
/*!*/;
# at 355
#200311 20:07:48 server id 1  end_log_pos 434 CRC32 0x123d65df  Anonymous_GTID last_committed=1 sequenc
# original_commit_timestamp=1583971668462467 (2020-03-11 20:07:48.462467 EDT)
# immediate_commit_timestamp=1583971668462467 (2020-03-11 20:07:48.462467 EDT)
/*!80001 SET @@session.original_commit_timestamp=1583971668462467*//*!*/;
/*!80014 SET @@session.original_server_version=80019*//*!*/;
/*!80014 SET @@session.immediate_server_version=80019*//*!*/;
SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
# at 434
#200311 20:07:48 server id 1  end_log_pos 828 CRC32 0x57fac9ac  Query thread_id=16 exec_time=0 error_co
use `pets`/*!*/;
SET TIMESTAMP=1583971668/*!*/;
/*!80013 SET @@session.sql_require_primary_key=0*//*!*/;
CREATE TABLE dogs
```

From the output of `mysqlbinlog`, the `DROP TABLE `pets`.`cats`` statement can be found in the segment of the binary log between the line `# at 232` and `# at 355`, which means the statement takes place *after* the log position 232, and the log is at position 355 after the `DROP TABLE` statement.

Note

Only use the `--start-datetime` and `--stop-datetime` options to help you find the actual event positions of interest. Using the two options to specify the range of binary log segment to apply is not recommended: there is a higher risk of missing binary log events when using the options. Use `--start-position` and `--stop-position` instead.

3. Apply the events in binary log file to the server, starting with the log position your found in step 1 (assume it is 155) and ending at the position you have found in step 2 that is *before* your point-in-time of interest (which is 232):

```
$> mysqlbinlog --start-position=155 --stop-position=232 /var/lib/mysql/bin.123456 \
        | mysql -u root -p
```

The command recovers all the transactions from the starting position until just before the stop position. Because the output of `mysqlbinlog` includes `SET TIMESTAMP` statements before each

SQL statement recorded, the recovered data and related MySQL logs reflect the original times at which the transactions were executed.

Your database has now been restored to the point-in-time of interest, t_p, right before the table `pets.cats` was dropped.

4. Beyond the point-in-time recovery that has been finished, if you also want to reexecute all the statements *after* your point-in-time of interest, use `mysqlbinlog` again to apply all the events after t_p to the server. We noted in step 2 that after the statement we wanted to skip, the log is at position 355; we can use it for the `--start-position` option, so that any statements after the position are included:

```
$> mysqlbinlog --start-position=355 /var/lib/mysql/bin.123456 \
        | mysql -u root -p
```

Your database has been restored the latest statement recorded in the binary log file, but with the selected event skipped.

7.6 MyISAM Table Maintenance and Crash Recovery

This section discusses how to use `myisamchk` to check or repair `MyISAM` tables (tables that have `.MYD` and `.MYI` files for storing data and indexes). For general `myisamchk` background, see Section 4.6.4, “myisamchk — MyISAM Table-Maintenance Utility”. Other table-repair information can be found at Section 2.10.13, “Rebuilding or Repairing Tables or Indexes”.

You can use `myisamchk` to check, repair, or optimize database tables. The following sections describe how to perform these operations and how to set up a table maintenance schedule. For information about using `myisamchk` to get information about your tables, see Section 4.6.4.5, “Obtaining Table Information with myisamchk”.

Even though table repair with `myisamchk` is quite secure, it is always a good idea to make a backup *before* doing a repair or any maintenance operation that could make a lot of changes to a table.

`myisamchk` operations that affect indexes can cause `MyISAM` `FULLTEXT` indexes to be rebuilt with full-text parameters that are incompatible with the values used by the MySQL server. To avoid this problem, follow the guidelines in Section 4.6.4.1, “myisamchk General Options”.

`MyISAM` table maintenance can also be done using the SQL statements that perform operations similar to what `myisamchk` can do:

- To check `MyISAM` tables, use `CHECK TABLE`.
- To repair `MyISAM` tables, use `REPAIR TABLE`.
- To optimize `MyISAM` tables, use `OPTIMIZE TABLE`.
- To analyze `MyISAM` tables, use `ANALYZE TABLE`.

For additional information about these statements, see Section 13.7.3, “Table Maintenance Statements”.

These statements can be used directly or by means of the `mysqlcheck` client program. One advantage of these statements over `myisamchk` is that the server does all the work. With `myisamchk`, you must make sure that the server does not use the tables at the same time so that there is no unwanted interaction between `myisamchk` and the server.

7.6.1 Using myisamchk for Crash Recovery

This section describes how to check for and deal with data corruption in MySQL databases. If your tables become corrupted frequently, you should try to find the reason why. See Section B.3.3.3, “What to Do If MySQL Keeps Crashing”.

For an explanation of how MyISAM tables can become corrupted, see Section 16.2.4, “MyISAM Table Problems”.

If you run mysqld with external locking disabled (which is the default), you cannot reliably use myisamchk to check a table when mysqld is using the same table. If you can be certain that no one can access the tables using mysqld while you run myisamchk, you only have to execute mysqladmin flush-tables before you start checking the tables. If you cannot guarantee this, you must stop mysqld while you check the tables. If you run myisamchk to check tables that mysqld is updating at the same time, you may get a warning that a table is corrupt even when it is not.

If the server is run with external locking enabled, you can use myisamchk to check tables at any time. In this case, if the server tries to update a table that myisamchk is using, the server waits for myisamchk to finish before it continues.

If you use myisamchk to repair or optimize tables, you *must* always ensure that the mysqld server is not using the table (this also applies if external locking is disabled). If you do not stop mysqld, you should at least do a mysqladmin flush-tables before you run myisamchk. Your tables *may become corrupted* if the server and myisamchk access the tables simultaneously.

When performing crash recovery, it is important to understand that each MyISAM table *tbl_name* in a database corresponds to the three files in the database directory shown in the following table.

File	Purpose
tbl_name.MYD	Data file
tbl_name.MYI	Index file

Each of these three file types is subject to corruption in various ways, but problems occur most often in data files and index files.

myisamchk works by creating a copy of the .MYD data file row by row. It ends the repair stage by removing the old .MYD file and renaming the new file to the original file name. If you use --quick, myisamchk does not create a temporary .MYD file, but instead assumes that the .MYD file is correct and generates only a new index file without touching the .MYD file. This is safe, because myisamchk automatically detects whether the .MYD file is corrupt and aborts the repair if it is. You can also specify the --quick option twice to myisamchk. In this case, myisamchk does not abort on some errors (such as duplicate-key errors) but instead tries to resolve them by modifying the .MYD file. Normally the use of two --quick options is useful only if you have too little free disk space to perform a normal repair. In this case, you should at least make a backup of the table before running myisamchk.

7.6.2 How to Check MyISAM Tables for Errors

To check a MyISAM table, use the following commands:

- myisamchk *tbl_name*

  This finds 99.99% of all errors. What it cannot find is corruption that involves *only* the data file (which is very unusual). If you want to check a table, you should normally run myisamchk without options or with the -s (silent) option.

- myisamchk -m *tbl_name*

  This finds 99.999% of all errors. It first checks all index entries for errors and then reads through all rows. It calculates a checksum for all key values in the rows and verifies that the checksum matches the checksum for the keys in the index tree.

- myisamchk -e *tbl_name*

  This does a complete and thorough check of all data (-e means “extended check”). It does a check-read of every key for each row to verify that they indeed point to the correct row. This may take a

long time for a large table that has many indexes. Normally, `myisamchk` stops after the first error it finds. If you want to obtain more information, you can add the `-v` (verbose) option. This causes `myisamchk` to keep going, up through a maximum of 20 errors.

- `myisamchk -e -i tbl_name`

  This is like the previous command, but the `-i` option tells `myisamchk` to print additional statistical information.

In most cases, a simple `myisamchk` command with no arguments other than the table name is sufficient to check a table.

7.6.3 How to Repair MyISAM Tables

The discussion in this section describes how to use `myisamchk` on `MyISAM` tables (extensions `.MYI` and `.MYD`).

You can also use the `CHECK TABLE` and `REPAIR TABLE` statements to check and repair `MyISAM` tables. See Section 13.7.3.2, “CHECK TABLE Statement”, and Section 13.7.3.5, “REPAIR TABLE Statement”.

Symptoms of corrupted tables include queries that abort unexpectedly and observable errors such as these:

- Can't find file *`tbl_name`*`.MYI` (Errcode: *`nnn`*)
- Unexpected end of file
- Record file is crashed
- Got error *`nnn`* from table handler

To get more information about the error, run `perror` *`nnn`*, where *`nnn`* is the error number. The following example shows how to use `perror` to find the meanings for the most common error numbers that indicate a problem with a table:

```
$> perror 126 127 132 134 135 136 141 144 145
MySQL error code 126 = Index file is crashed
MySQL error code 127 = Record-file is crashed
MySQL error code 132 = Old database file
MySQL error code 134 = Record was already deleted (or record file crashed)
MySQL error code 135 = No more room in record file
MySQL error code 136 = No more room in index file
MySQL error code 141 = Duplicate unique key or constraint on write or update
MySQL error code 144 = Table is crashed and last repair failed
MySQL error code 145 = Table was marked as crashed and should be repaired
```

Note that error 135 (no more room in record file) and error 136 (no more room in index file) are not errors that can be fixed by a simple repair. In this case, you must use `ALTER TABLE` to increase the `MAX_ROWS` and `AVG_ROW_LENGTH` table option values:

```
ALTER TABLE tbl_name MAX_ROWS=xxx AVG_ROW_LENGTH=yyy;
```

If you do not know the current table option values, use `SHOW CREATE TABLE`.

For the other errors, you must repair your tables. `myisamchk` can usually detect and fix most problems that occur.

The repair process involves up to three stages, described here. Before you begin, you should change location to the database directory and check the permissions of the table files. On Unix, make sure that they are readable by the user that `mysqld` runs as (and to you, because you need to access the files you are checking). If it turns out you need to modify files, they must also be writable by you.

This section is for the cases where a table check fails (such as those described in Section 7.6.2, “How to Check MyISAM Tables for Errors”), or you want to use the extended features that `myisamchk` provides.

The `myisamchk` options used for table maintenance with are described in Section 4.6.4, “myisamchk — MyISAM Table-Maintenance Utility”. `myisamchk` also has variables that you can set to control memory allocation that may improve performance. See Section 4.6.4.6, “myisamchk Memory Usage”.

If you are going to repair a table from the command line, you must first stop the `mysqld` server. Note that when you do `mysqladmin shutdown` on a remote server, the `mysqld` server is still available for a while after `mysqladmin` returns, until all statement-processing has stopped and all index changes have been flushed to disk.

Stage 1: Checking your tables

Run `myisamchk *.MYI` or `myisamchk -e *.MYI` if you have more time. Use the `-s` (silent) option to suppress unnecessary information.

If the `mysqld` server is stopped, you should use the `--update-state` option to tell `myisamchk` to mark the table as “checked.”

You have to repair only those tables for which `myisamchk` announces an error. For such tables, proceed to Stage 2.

If you get unexpected errors when checking (such as `out of memory` errors), or if `myisamchk` crashes, go to Stage 3.

Stage 2: Easy safe repair

First, try `myisamchk -r -q tbl_name` (`-r -q` means “quick recovery mode”). This attempts to repair the index file without touching the data file. If the data file contains everything that it should and the delete links point at the correct locations within the data file, this should work, and the table is fixed. Start repairing the next table. Otherwise, use the following procedure:

1. Make a backup of the data file before continuing.
2. Use `myisamchk -r tbl_name` (`-r` means “recovery mode”). This removes incorrect rows and deleted rows from the data file and reconstructs the index file.
3. If the preceding step fails, use `myisamchk --safe-recover tbl_name`. Safe recovery mode uses an old recovery method that handles a few cases that regular recovery mode does not (but is slower).

Note

If you want a repair operation to go much faster, you should set the values of the `sort_buffer_size` and `key_buffer_size` variables each to about 25% of your available memory when running `myisamchk`.

If you get unexpected errors when repairing (such as `out of memory` errors), or if `myisamchk` crashes, go to Stage 3.

Stage 3: Difficult repair

You should reach this stage only if the first 16KB block in the index file is destroyed or contains incorrect information, or if the index file is missing. In this case, it is necessary to create a new index file. Do so as follows:

1. Move the data file to a safe place.
2. Use the table description file to create new (empty) data and index files:

```
$> mysql db_name
```

```
mysql> SET autocommit=1;
mysql> TRUNCATE TABLE tbl_name;
mysql> quit
```

3. Copy the old data file back onto the newly created data file. (Do not just move the old file back onto the new file. You want to retain a copy in case something goes wrong.)

Important

If you are using replication, you should stop it prior to performing the above procedure, since it involves file system operations, and these are not logged by MySQL.

Go back to Stage 2. `myisamchk -r -q` should work. (This should not be an endless loop.)

You can also use the `REPAIR TABLE tbl_name USE_FRM` SQL statement, which performs the whole procedure automatically. There is also no possibility of unwanted interaction between a utility and the server, because the server does all the work when you use `REPAIR TABLE`. See Section 13.7.3.5, “REPAIR TABLE Statement”.

7.6.4 MyISAM Table Optimization

To coalesce fragmented rows and eliminate wasted space that results from deleting or updating rows, run `myisamchk` in recovery mode:

```
$> myisamchk -r tbl_name
```

You can optimize a table in the same way by using the `OPTIMIZE TABLE` SQL statement. `OPTIMIZE TABLE` does a table repair and a key analysis, and also sorts the index tree so that key lookups are faster. There is also no possibility of unwanted interaction between a utility and the server, because the server does all the work when you use `OPTIMIZE TABLE`. See Section 13.7.3.4, “OPTIMIZE TABLE Statement”.

`myisamchk` has a number of other options that you can use to improve the performance of a table:

- `--analyze` or `-a`: Perform key distribution analysis. This improves join performance by enabling the join optimizer to better choose the order in which to join the tables and which indexes it should use.
- `--sort-index` or `-S`: Sort the index blocks. This optimizes seeks and makes table scans that use indexes faster.
- `--sort-records=index_num` or `-R index_num`: Sort data rows according to a given index. This makes your data much more localized and may speed up range-based `SELECT` and `ORDER BY` operations that use this index.

For a full description of all available options, see Section 4.6.4, “myisamchk — MyISAM Table-Maintenance Utility”.

7.6.5 Setting Up a MyISAM Table Maintenance Schedule

It is a good idea to perform table checks on a regular basis rather than waiting for problems to occur. One way to check and repair `MyISAM` tables is with the `CHECK TABLE` and `REPAIR TABLE` statements. See Section 13.7.3, “Table Maintenance Statements”.

Another way to check tables is to use `myisamchk`. For maintenance purposes, you can use `myisamchk -s`. The `-s` option (short for `--silent`) causes `myisamchk` to run in silent mode, printing messages only when errors occur.

It is also a good idea to enable automatic `MyISAM` table checking. For example, whenever the machine has done a restart in the middle of an update, you usually need to check each table that could have

been affected before it is used further. (These are “expected crashed tables.”) To cause the server to check MyISAM tables automatically, start it with the myisam_recover_options system variable set. See Section 5.1.8, “Server System Variables”.

You should also check your tables regularly during normal system operation. For example, you can run a cron job to check important tables once a week, using a line like this in a crontab file:

```
35 0 * * 0 /path/to/myisamchk --fast --silent /path/to/datadir/*/*.MYI
```

This prints out information about crashed tables so that you can examine and repair them as necessary.

To start with, execute myisamchk -s each night on all tables that have been updated during the last 24 hours. As you see that problems occur infrequently, you can back off the checking frequency to once a week or so.

Normally, MySQL tables need little maintenance. If you are performing many updates to MyISAM tables with dynamic-sized rows (tables with VARCHAR, BLOB, or TEXT columns) or have tables with many deleted rows you may want to defragment/reclaim space from the tables from time to time. You can do this by using OPTIMIZE TABLE on the tables in question. Alternatively, if you can stop the mysqld server for a while, change location into the data directory and use this command while the server is stopped:

```
$> myisamchk -r -s --sort-index --myisam_sort_buffer_size=16M */*.MYI
```

Chapter 8 Optimization

Table of Contents

This chapter explains how to optimize MySQL performance and provides examples. Optimization involves configuring, tuning, and measuring performance, at several levels. Depending on your job role (developer, DBA, or a combination of both), you might optimize at the level of individual SQL statements, entire applications, a single database server, or multiple networked database servers. Sometimes you can be proactive and plan in advance for performance, while other times you might troubleshoot a configuration or code issue after a problem occurs. Optimizing CPU and memory usage can also improve scalability, allowing the database to handle more load without slowing down.

8.1 Optimization Overview

Database performance depends on several factors at the database level, such as tables, queries, and configuration settings. These software constructs result in CPU and I/O operations at the hardware level, which you must minimize and make as efficient as possible. As you work on database performance, you start by learning the high-level rules and guidelines for the software side, and measuring performance using wall-clock time. As you become an expert, you learn more about what happens internally, and start measuring things such as CPU cycles and I/O operations.

Typical users aim to get the best database performance out of their existing software and hardware configurations. Advanced users look for opportunities to improve the MySQL software itself, or develop their own storage engines and hardware appliances to expand the MySQL ecosystem.

Optimizing at the Database Level

The most important factor in making a database application fast is its basic design:

- Are the tables structured properly? In particular, do the columns have the right data types, and does each table have the appropriate columns for the type of work? For example, applications that perform frequent updates often have many tables with few columns, while applications that analyze large amounts of data often have few tables with many columns.
- Are the right indexes in place to make queries efficient?
- Are you using the appropriate storage engine for each table, and taking advantage of the strengths and features of each storage engine you use? In particular, the choice of a transactional storage engine such as `InnoDB` or a nontransactional one such as `MyISAM` can be very important for performance and scalability.

Note

`InnoDB` is the default storage engine for new tables. In practice, the advanced `InnoDB` performance features mean that `InnoDB` tables often outperform the simpler `MyISAM` tables, especially for a busy database.

- Does each table use an appropriate row format? This choice also depends on the storage engine used for the table. In particular, compressed tables use less disk space and so require less disk I/O to read and write the data. Compression is available for all kinds of workloads with `InnoDB` tables, and for read-only `MyISAM` tables.
- Does the application use an appropriate locking strategy? For example, by allowing shared access when possible so that database operations can run concurrently, and requesting exclusive access when appropriate so that critical operations get top priority. Again, the choice of storage engine is significant. The `InnoDB` storage engine handles most locking issues without involvement from you, allowing for better concurrency in the database and reducing the amount of experimentation and tuning for your code.
- Are all memory areas used for caching sized correctly? That is, large enough to hold frequently accessed data, but not so large that they overload physical memory and cause paging. The main memory areas to configure are the `InnoDB` buffer pool and the `MyISAM` key cache.

Optimizing at the Hardware Level

Any database application eventually hits hardware limits as the database becomes more and more busy. A DBA must evaluate whether it is possible to tune the application or reconfigure the server to avoid these bottlenecks, or whether more hardware resources are required. System bottlenecks typically arise from these sources:

- Disk seeks. It takes time for the disk to find a piece of data. With modern disks, the mean time for this is usually lower than 10ms, so we can in theory do about 100 seeks a second. This time improves slowly with new disks and is very hard to optimize for a single table. The way to optimize seek time is to distribute the data onto more than one disk.

- Disk reading and writing. When the disk is at the correct position, we need to read or write the data. With modern disks, one disk delivers at least 10–20MB/s throughput. This is easier to optimize than seeks because you can read in parallel from multiple disks.
- CPU cycles. When the data is in main memory, we must process it to get our result. Having large tables compared to the amount of memory is the most common limiting factor. But with small tables, speed is usually not the problem.
- Memory bandwidth. When the CPU needs more data than can fit in the CPU cache, main memory bandwidth becomes a bottleneck. This is an uncommon bottleneck for most systems, but one to be aware of.

Balancing Portability and Performance

To use performance-oriented SQL extensions in a portable MySQL program, you can wrap MySQL-specific keywords in a statement within `/*! */` comment delimiters. Other SQL servers ignore the commented keywords. For information about writing comments, see Section 9.7, “Comments”.

8.2 Optimizing SQL Statements

The core logic of a database application is performed through SQL statements, whether issued directly through an interpreter or submitted behind the scenes through an API. The tuning guidelines in this section help to speed up all kinds of MySQL applications. The guidelines cover SQL operations that read and write data, the behind-the-scenes overhead for SQL operations in general, and operations used in specific scenarios such as database monitoring.

8.2.1 Optimizing SELECT Statements

Queries, in the form of `SELECT` statements, perform all the lookup operations in the database. Tuning these statements is a top priority, whether to achieve sub-second response times for dynamic web pages, or to chop hours off the time to generate huge overnight reports.

Besides `SELECT` statements, the tuning techniques for queries also apply to constructs such as `CREATE TABLE...AS SELECT`, `INSERT INTO...SELECT`, and `WHERE` clauses in `DELETE` statements. Those statements have additional performance considerations because they combine write operations with the read-oriented query operations.

NDB Cluster supports a join pushdown optimization whereby a qualifying join is sent in its entirety to NDB Cluster data nodes, where it can be distributed among them and executed in parallel. For more information about this optimization, see Conditions for NDB pushdown joins.

The main considerations for optimizing queries are:

- To make a slow `SELECT ... WHERE` query faster, the first thing to check is whether you can add an index. Set up indexes on columns used in the `WHERE` clause, to speed up evaluation, filtering, and the final retrieval of results. To avoid wasted disk space, construct a small set of indexes that speed up many related queries used in your application.

  Indexes are especially important for queries that reference different tables, using features such as joins and foreign keys. You can use the `EXPLAIN` statement to determine which indexes are used for a `SELECT`. See Section 8.3.1, “How MySQL Uses Indexes” and Section 8.8.1, “Optimizing Queries with EXPLAIN”.

- Isolate and tune any part of the query, such as a function call, that takes excessive time. Depending on how the query is structured, a function could be called once for every row in the result set, or even once for every row in the table, greatly magnifying any inefficiency.
- Minimize the number of full table scans in your queries, particularly for big tables.
- Keep table statistics up to date by using the `ANALYZE TABLE` statement periodically, so the optimizer has the information needed to construct an efficient execution plan.

- Learn the tuning techniques, indexing techniques, and configuration parameters that are specific to the storage engine for each table. Both `InnoDB` and `MyISAM` have sets of guidelines for enabling and sustaining high performance in queries. For details, see Section 8.5.6, “Optimizing InnoDB Queries” and Section 8.6.1, “Optimizing MyISAM Queries”.
- You can optimize single-query transactions for `InnoDB` tables, using the technique in Section 8.5.3, “Optimizing InnoDB Read-Only Transactions”.
- Avoid transforming the query in ways that make it hard to understand, especially if the optimizer does some of the same transformations automatically.
- If a performance issue is not easily solved by one of the basic guidelines, investigate the internal details of the specific query by reading the `EXPLAIN` plan and adjusting your indexes, `WHERE` clauses, join clauses, and so on. (When you reach a certain level of expertise, reading the `EXPLAIN` plan might be your first step for every query.)
- Adjust the size and properties of the memory areas that MySQL uses for caching. With efficient use of the `InnoDB` buffer pool, `MyISAM` key cache, and the MySQL query cache, repeated queries run faster because the results are retrieved from memory the second and subsequent times.
- Even for a query that runs fast using the cache memory areas, you might still optimize further so that they require less cache memory, making your application more scalable. Scalability means that your application can handle more simultaneous users, larger requests, and so on without experiencing a big drop in performance.
- Deal with locking issues, where the speed of your query might be affected by other sessions accessing the tables at the same time.

8.2.1.1 WHERE Clause Optimization

This section discusses optimizations that can be made for processing `WHERE` clauses. The examples use `SELECT` statements, but the same optimizations apply for `WHERE` clauses in `DELETE` and `UPDATE` statements.

Note

Because work on the MySQL optimizer is ongoing, not all of the optimizations that MySQL performs are documented here.

You might be tempted to rewrite your queries to make arithmetic operations faster, while sacrificing readability. Because MySQL does similar optimizations automatically, you can often avoid this work, and leave the query in a more understandable and maintainable form. Some of the optimizations performed by MySQL follow:

- Removal of unnecessary parentheses:

```
   ((a AND b) AND c OR (((a AND b) AND (c AND d))))
-> (a AND b AND c) OR (a AND b AND c AND d)
```

- Constant folding:

```
   (a<b AND b=c) AND a=5
-> b>5 AND b=c AND a=5
```

- Constant condition removal:

```
   (b>=5 AND b=5) OR (b=6 AND 5=5) OR (b=7 AND 5=6)
-> b=5 OR b=6
```

In MySQL 8.0.14 and later, this takes place during preparation rather than during the optimization phase, which helps in simplification of joins. See Section 8.2.1.9, “Outer Join Optimization”, for further information and examples.

- Constant expressions used by indexes are evaluated only once.
- Beginning with MySQL 8.0.16, comparisons of columns of numeric types with constant values are checked and folded or removed for invalid or out-of-rage values:

```
# CREATE TABLE t (c TINYINT UNSIGNED NOT NULL);
  SELECT * FROM t WHERE c ≪ 256;
-≫ SELECT * FROM t WHERE 1;
```

  See Section 8.2.1.14, “Constant-Folding Optimization”, for more information.

- `COUNT(*)` on a single table without a `WHERE` is retrieved directly from the table information for `MyISAM` and `MEMORY` tables. This is also done for any `NOT NULL` expression when used with only one table.
- Early detection of invalid constant expressions. MySQL quickly detects that some `SELECT` statements are impossible and returns no rows.
- `HAVING` is merged with `WHERE` if you do not use `GROUP BY` or aggregate functions (`COUNT()`, `MIN()`, and so on).
- For each table in a join, a simpler `WHERE` is constructed to get a fast `WHERE` evaluation for the table and also to skip rows as soon as possible.
- All constant tables are read first before any other tables in the query. A constant table is any of the following:
  - An empty table or a table with one row.
  - A table that is used with a `WHERE` clause on a `PRIMARY KEY` or a `UNIQUE` index, where all index parts are compared to constant expressions and are defined as `NOT NULL`.

  All of the following tables are used as constant tables:

```
SELECT * FROM t WHERE primary_key=1;
SELECT * FROM t1,t2
  WHERE t1.primary_key=1 AND t2.primary_key=t1.id;
```

- The best join combination for joining the tables is found by trying all possibilities. If all columns in `ORDER BY` and `GROUP BY` clauses come from the same table, that table is preferred first when joining.
- If there is an `ORDER BY` clause and a different `GROUP BY` clause, or if the `ORDER BY` or `GROUP BY` contains columns from tables other than the first table in the join queue, a temporary table is created.
- If you use the `SQL_SMALL_RESULT` modifier, MySQL uses an in-memory temporary table.
- Each table index is queried, and the best index is used unless the optimizer believes that it is more efficient to use a table scan. At one time, a scan was used based on whether the best index spanned more than 30% of the table, but a fixed percentage no longer determines the choice between using an index or a scan. The optimizer now is more complex and bases its estimate on additional factors such as table size, number of rows, and I/O block size.
- In some cases, MySQL can read rows from the index without even consulting the data file. If all columns used from the index are numeric, only the index tree is used to resolve the query.
- Before each row is output, those that do not match the `HAVING` clause are skipped.

Some examples of queries that are very fast:

```
SELECT COUNT(*) FROM tbl_name;

SELECT MIN(key_part1),MAX(key_part1) FROM tbl_name;
```

```
SELECT MAX(key_part2) FROM tbl_name
  WHERE key_part1=constant;

SELECT ... FROM tbl_name
  ORDER BY key_part1,key_part2,... LIMIT 10;

SELECT ... FROM tbl_name
  ORDER BY key_part1 DESC, key_part2 DESC, ... LIMIT 10;
```

MySQL resolves the following queries using only the index tree, assuming that the indexed columns are numeric:

```
SELECT key_part1,key_part2 FROM tbl_name WHERE key_part1=val;

SELECT COUNT(*) FROM tbl_name
  WHERE key_part1=val1 AND key_part2=val2;

SELECT MAX(key_part2) FROM tbl_name GROUP BY key_part1;
```

The following queries use indexing to retrieve the rows in sorted order without a separate sorting pass:

```
SELECT ... FROM tbl_name
  ORDER BY key_part1,key_part2,... ;

SELECT ... FROM tbl_name
  ORDER BY key_part1 DESC, key_part2 DESC, ... ;
```

8.2.1.2 Range Optimization

The `range` access method uses a single index to retrieve a subset of table rows that are contained within one or several index value intervals. It can be used for a single-part or multiple-part index. The following sections describe conditions under which the optimizer uses range access.

- Range Access Method for Single-Part Indexes
- Range Access Method for Multiple-Part Indexes
- Equality Range Optimization of Many-Valued Comparisons
- Skip Scan Range Access Method
- Range Optimization of Row Constructor Expressions
- Limiting Memory Use for Range Optimization

Range Access Method for Single-Part Indexes

For a single-part index, index value intervals can be conveniently represented by corresponding conditions in the `WHERE` clause, denoted as *range conditions* rather than “intervals.”

The definition of a range condition for a single-part index is as follows:

- For both `BTREE` and `HASH` indexes, comparison of a key part with a constant value is a range condition when using the `=`, `<=>`, `IN()`, `IS NULL`, or `IS NOT NULL` operators.
- Additionally, for `BTREE` indexes, comparison of a key part with a constant value is a range condition when using the `>`, `<`, `>=`, `<=`, `BETWEEN`, `!=`, or `<>` operators, or `LIKE` comparisons if the argument to `LIKE` is a constant string that does not start with a wildcard character.
- For all index types, multiple range conditions combined with `OR` or `AND` form a range condition.

“Constant value” in the preceding descriptions means one of the following:

- A constant from the query string
- A column of a `const` or `system` table from the same join

- The result of an uncorrelated subquery
- Any expression composed entirely from subexpressions of the preceding types

Here are some examples of queries with range conditions in the `WHERE` clause:

```
SELECT * FROM t1
  WHERE key_col > 1
  AND key_col < 10;

SELECT * FROM t1
  WHERE key_col = 1
  OR key_col IN (15,18,20);

SELECT * FROM t1
  WHERE key_col LIKE 'ab%'
  OR key_col BETWEEN 'bar' AND 'foo';
```

Some nonconstant values may be converted to constants during the optimizer constant propagation phase.

MySQL tries to extract range conditions from the `WHERE` clause for each of the possible indexes. During the extraction process, conditions that cannot be used for constructing the range condition are dropped, conditions that produce overlapping ranges are combined, and conditions that produce empty ranges are removed.

Consider the following statement, where `key1` is an indexed column and `nonkey` is not indexed:

```
SELECT * FROM t1 WHERE
  (key1 < 'abc' AND (key1 LIKE 'abcde%' OR key1 LIKE '%b')) OR
  (key1 < 'bar' AND nonkey = 4) OR
  (key1 < 'uux' AND key1 > 'z');
```

The extraction process for key `key1` is as follows:

1. Start with original `WHERE` clause:

   ```
   (key1 < 'abc' AND (key1 LIKE 'abcde%' OR key1 LIKE '%b')) OR
   (key1 < 'bar' AND nonkey = 4) OR
   (key1 < 'uux' AND key1 > 'z')
   ```

2. Remove `nonkey = 4` and `key1 LIKE '%b'` because they cannot be used for a range scan. The correct way to remove them is to replace them with `TRUE`, so that we do not miss any matching rows when doing the range scan. Replacing them with `TRUE` yields:

   ```
   (key1 < 'abc' AND (key1 LIKE 'abcde%' OR TRUE)) OR
   (key1 < 'bar' AND TRUE) OR
   (key1 < 'uux' AND key1 > 'z')
   ```

3. Collapse conditions that are always true or false:

   - `(key1 LIKE 'abcde%' OR TRUE)` is always true
   - `(key1 < 'uux' AND key1 > 'z')` is always false

   Replacing these conditions with constants yields:

   ```
   (key1 < 'abc' AND TRUE) OR (key1 < 'bar' AND TRUE) OR (FALSE)
   ```

   Removing unnecessary `TRUE` and `FALSE` constants yields:

   ```
   (key1 < 'abc') OR (key1 < 'bar')
   ```

4. Combining overlapping intervals into one yields the final condition to be used for the range scan:

   ```
   (key1 < 'bar')
   ```

In general (and as demonstrated by the preceding example), the condition used for a range scan is less restrictive than the `WHERE` clause. MySQL performs an additional check to filter out rows that satisfy the range condition but not the full `WHERE` clause.

The range condition extraction algorithm can handle nested `AND`/`OR` constructs of arbitrary depth, and its output does not depend on the order in which conditions appear in `WHERE` clause.

MySQL does not support merging multiple ranges for the `range` access method for spatial indexes. To work around this limitation, you can use a `UNION` with identical `SELECT` statements, except that you put each spatial predicate in a different `SELECT`.

Range Access Method for Multiple-Part Indexes

Range conditions on a multiple-part index are an extension of range conditions for a single-part index. A range condition on a multiple-part index restricts index rows to lie within one or several key tuple intervals. Key tuple intervals are defined over a set of key tuples, using ordering from the index.

For example, consider a multiple-part index defined as `key1(key_part1, key_part2, key_part3)`, and the following set of key tuples listed in key order:

```
key_part1  key_part2  key_part3
  NULL       1          'abc'
  NULL       1          'xyz'
  NULL       2          'foo'
   1         1          'abc'
   1         1          'xyz'
   1         2          'abc'
   2         1          'aaa'
```

The condition `key_part1 = 1` defines this interval:

```
(1,-inf,-inf) <= (key_part1,key_part2,key_part3) < (1,+inf,+inf)
```

The interval covers the 4th, 5th, and 6th tuples in the preceding data set and can be used by the range access method.

By contrast, the condition `key_part3 = 'abc'` does not define a single interval and cannot be used by the range access method.

The following descriptions indicate how range conditions work for multiple-part indexes in greater detail.

- For `HASH` indexes, each interval containing identical values can be used. This means that the interval can be produced only for conditions in the following form:

  ```
      key_part1 cmp const1
  AND key_part2 cmp const2
  AND ...
  AND key_partN cmp constN;
  ```

  Here, `const1`, `const2`, ... are constants, `cmp` is one of the `=`, `<=>`, or `IS NULL` comparison operators, and the conditions cover all index parts. (That is, there are `N` conditions, one for each part of an `N`-part index.) For example, the following is a range condition for a three-part `HASH` index:

  ```
  key_part1 = 1 AND key_part2 IS NULL AND key_part3 = 'foo'
  ```

  For the definition of what is considered to be a constant, see Range Access Method for Single-Part Indexes.

- For a `BTREE` index, an interval might be usable for conditions combined with `AND`, where each condition compares a key part with a constant value using `=`, `<=>`, `IS NULL`, `>`, `<`, `>=`, `<=`, `!=`, `<>`, `BETWEEN`, or `LIKE 'pattern'` (where `'pattern'` does not start with a wildcard). An interval can be used as long as it is possible to determine a single key tuple containing all rows that match the condition (or two intervals if `<>` or `!=` is used).

The optimizer attempts to use additional key parts to determine the interval as long as the comparison operator is `=`, `<=>`, or `IS NULL`. If the operator is `>`, `<`, `>=`, `<=`, `!=`, `<>`, `BETWEEN`, or `LIKE`, the optimizer uses it but considers no more key parts. For the following expression, the optimizer uses `=` from the first comparison. It also uses `>=` from the second comparison but considers no further key parts and does not use the third comparison for interval construction:

```
key_part1 = 'foo' AND key_part2 >= 10 AND key_part3 > 10
```

The single interval is:

```
('foo',10,-inf) < (key_part1,key_part2,key_part3) < ('foo',+inf,+inf)
```

It is possible that the created interval contains more rows than the initial condition. For example, the preceding interval includes the value `('foo', 11, 0)`, which does not satisfy the original condition.

- If conditions that cover sets of rows contained within intervals are combined with `OR`, they form a condition that covers a set of rows contained within the union of their intervals. If the conditions are combined with `AND`, they form a condition that covers a set of rows contained within the intersection of their intervals. For example, for this condition on a two-part index:

  ```
  (key_part1 = 1 AND key_part2 < 2) OR (key_part1 > 5)
  ```

  The intervals are:

  ```
  (1,-inf) < (key_part1,key_part2) < (1,2)
  (5,-inf) < (key_part1,key_part2)
  ```

  In this example, the interval on the first line uses one key part for the left bound and two key parts for the right bound. The interval on the second line uses only one key part. The `key_len` column in the `EXPLAIN` output indicates the maximum length of the key prefix used.

  In some cases, `key_len` may indicate that a key part was used, but that might be not what you would expect. Suppose that *`key_part1`* and *`key_part2`* can be `NULL`. Then the `key_len` column displays two key part lengths for the following condition:

  ```
  key_part1 >= 1 AND key_part2 < 2
  ```

  But, in fact, the condition is converted to this:

  ```
  key_part1 >= 1 AND key_part2 IS NOT NULL
  ```

For a description of how optimizations are performed to combine or eliminate intervals for range conditions on a single-part index, see Range Access Method for Single-Part Indexes. Analogous steps are performed for range conditions on multiple-part indexes.

Equality Range Optimization of Many-Valued Comparisons

Consider these expressions, where *`col_name`* is an indexed column:

```
col_name IN(val1, ..., valN)
col_name = val1 OR ... OR col_name = valN
```

Each expression is true if *`col_name`* is equal to any of several values. These comparisons are equality range comparisons (where the “range” is a single value). The optimizer estimates the cost of reading qualifying rows for equality range comparisons as follows:

- If there is a unique index on *`col_name`*, the row estimate for each range is 1 because at most one row can have the given value.
- Otherwise, any index on *`col_name`* is nonunique and the optimizer can estimate the row count for each range using dives into the index or index statistics.

With index dives, the optimizer makes a dive at each end of a range and uses the number of rows in the range as the estimate. For example, the expression *`col_name`* `IN (10, 20, 30)` has three equality ranges and the optimizer makes two dives per range to generate a row estimate. Each pair of dives yields an estimate of the number of rows that have the given value.

Index dives provide accurate row estimates, but as the number of comparison values in the expression increases, the optimizer takes longer to generate a row estimate. Use of index statistics is less accurate than index dives but permits faster row estimation for large value lists.

The `eq_range_index_dive_limit` system variable enables you to configure the number of values at which the optimizer switches from one row estimation strategy to the other. To permit use of index dives for comparisons of up to *`N`* equality ranges, set `eq_range_index_dive_limit` to *`N`* + 1. To disable use of statistics and always use index dives regardless of *`N`*, set `eq_range_index_dive_limit` to 0.

To update table index statistics for best estimates, use `ANALYZE TABLE`.

Prior to MySQL 8.0, there is no way of skipping the use of index dives to estimate index usefulness, except by using the `eq_range_index_dive_limit` system variable. In MySQL 8.0, index dive skipping is possible for queries that satisfy all these conditions:

- The query is for a single table, not a join on multiple tables.
- A single-index `FORCE INDEX` index hint is present. The idea is that if index use is forced, there is nothing to be gained from the additional overhead of performing dives into the index.
- The index is nonunique and not a `FULLTEXT` index.
- No subquery is present.
- No `DISTINCT`, `GROUP BY`, or `ORDER BY` clause is present.

For `EXPLAIN FOR CONNECTION`, the output changes as follows if index dives are skipped:

- For traditional output, the `rows` and `filtered` values are `NULL`.
- For JSON output, `rows_examined_per_scan` and `rows_produced_per_join` do not appear, `skip_index_dive_due_to_force` is `true`, and cost calculations are not accurate.

Without `FOR CONNECTION`, `EXPLAIN` output does not change when index dives are skipped.

After execution of a query for which index dives are skipped, the corresponding row in the Information Schema `OPTIMIZER_TRACE` table contains an `index_dives_for_range_access` value of `skipped_due_to_force_index`.

Skip Scan Range Access Method

Consider the following scenario:

```
CREATE TABLE t1 (f1 INT NOT NULL, f2 INT NOT NULL, PRIMARY KEY(f1, f2));
INSERT INTO t1 VALUES
  (1,1), (1,2), (1,3), (1,4), (1,5),
  (2,1), (2,2), (2,3), (2,4), (2,5);
INSERT INTO t1 SELECT f1, f2 + 5 FROM t1;
INSERT INTO t1 SELECT f1, f2 + 10 FROM t1;
INSERT INTO t1 SELECT f1, f2 + 20 FROM t1;
INSERT INTO t1 SELECT f1, f2 + 40 FROM t1;
ANALYZE TABLE t1;

EXPLAIN SELECT f1, f2 FROM t1 WHERE f2 > 40;
```

To execute this query, MySQL can choose an index scan to fetch all rows (the index includes all columns to be selected), then apply the `f2 > 40` condition from the `WHERE` clause to produce the final result set.

A range scan is more efficient than a full index scan, but cannot be used in this case because there is no condition on `f1`, the first index column. However, as of MySQL 8.0.13, the optimizer can perform multiple range scans, one for each value of `f1`, using a method called Skip Scan that is similar to Loose Index Scan (see Section 8.2.1.17, “GROUP BY Optimization”):

1. Skip between distinct values of the first index part, `f1` (the index prefix).
2. Perform a subrange scan on each distinct prefix value for the `f2 > 40` condition on the remaining index part.

For the data set shown earlier, the algorithm operates like this:

1. Get the first distinct value of the first key part (`f1 = 1`).
2. Construct the range based on the first and second key parts (`f1 = 1 AND f2 > 40`).
3. Perform a range scan.
4. Get the next distinct value of the first key part (`f1 = 2`).
5. Construct the range based on the first and second key parts (`f1 = 2 AND f2 > 40`).
6. Perform a range scan.

Using this strategy decreases the number of accessed rows because MySQL skips the rows that do not qualify for each constructed range. This Skip Scan access method is applicable under the following conditions:

- Table T has at least one compound index with key parts of the form ([A_1, ..., A_*k*,] B_1, ..., B_*m*, C [, D_1, ..., D_*n*]). Key parts A and D may be empty, but B and C must be nonempty.
- The query references only one table.
- The query does not use `GROUP BY` or `DISTINCT`.
- The query references only columns in the index.
- The predicates on A_1, ..., A_*k* must be equality predicates and they must be constants. This includes the `IN()` operator.
- The query must be a conjunctive query; that is, an `AND` of `OR` conditions: `(`*`cond1`*`(`*`key_part1`*`) OR `*`cond2`*`(`*`key_part1`*`)) AND (`*`cond1`*`(`*`key_part2`*`) OR ...) AND ...`
- There must be a range condition on C.
- Conditions on D columns are permitted. Conditions on D must be in conjunction with the range condition on C.

Use of Skip Scan is indicated in `EXPLAIN` output as follows:

- `Using index for skip scan` in the `Extra` column indicates that the loose index Skip Scan access method is used.
- If the index can be used for Skip Scan, the index should be visible in the `possible_keys` column.

Use of Skip Scan is indicated in optimizer trace output by a `"skip scan"` element of this form:

```
"skip_scan_range": {
  "type": "skip_scan",
  "index": index_used_for_skip_scan,
  "key_parts_used_for_access": [key_parts_used_for_access],
  "range": [range]
}
```

You may also see a `"best_skip_scan_summary"` element. If Skip Scan is chosen as the best range access variant, a `"chosen_range_access_summary"` is written. If Skip Scan is chosen as the overall best access method, a `"best_access_path"` element is present.

Use of Skip Scan is subject to the value of the `skip_scan` flag of the `optimizer_switch` system variable. See Section 8.9.2, “Switchable Optimizations”. By default, this flag is `on`. To disable it, set `skip_scan` to `off`.

In addition to using the `optimizer_switch` system variable to control optimizer use of Skip Scan session-wide, MySQL supports optimizer hints to influence the optimizer on a per-statement basis. See Section 8.9.3, “Optimizer Hints”.

Range Optimization of Row Constructor Expressions

The optimizer is able to apply the range scan access method to queries of this form:

```
SELECT ... FROM t1 WHERE ( col_1, col_2 ) IN (( 'a', 'b' ), ( 'c', 'd' ));
```

Previously, for range scans to be used, it was necessary to write the query as:

```
SELECT ... FROM t1 WHERE ( col_1 = 'a' AND col_2 = 'b' )
OR ( col_1 = 'c' AND col_2 = 'd' );
```

For the optimizer to use a range scan, queries must satisfy these conditions:

- Only `IN()` predicates are used, not `NOT IN()`.
- On the left side of the `IN()` predicate, the row constructor contains only column references.
- On the right side of the `IN()` predicate, row constructors contain only runtime constants, which are either literals or local column references that are bound to constants during execution.
- On the right side of the `IN()` predicate, there is more than one row constructor.

For more information about the optimizer and row constructors, see Section 8.2.1.22, “Row Constructor Expression Optimization”

Limiting Memory Use for Range Optimization

To control the memory available to the range optimizer, use the `range_optimizer_max_mem_size` system variable:

- A value of 0 means “no limit.”
- With a value greater than 0, the optimizer tracks the memory consumed when considering the range access method. If the specified limit is about to be exceeded, the range access method is abandoned and other methods, including a full table scan, are considered instead. This could be less optimal. If this happens, the following warning occurs (where *`N`* is the current `range_optimizer_max_mem_size` value):

  ```
  Warning    3170    Memory capacity of N bytes for
                     'range_optimizer_max_mem_size' exceeded. Range
                     optimization was not done for this query.
  ```

- For `UPDATE` and `DELETE` statements, if the optimizer falls back to a full table scan and the `sql_safe_updates` system variable is enabled, an error occurs rather than a warning because, in effect, no key is used to determine which rows to modify. For more information, see Using Safe-Updates Mode (--safe-updates).

For individual queries that exceed the available range optimization memory and for which the optimizer falls back to less optimal plans, increasing the `range_optimizer_max_mem_size` value may improve performance.

To estimate the amount of memory needed to process a range expression, use these guidelines:

- For a simple query such as the following, where there is one candidate key for the range access method, each predicate combined with `OR` uses approximately 230 bytes:

```
SELECT COUNT(*) FROM t
WHERE a=1 OR a=2 OR a=3 OR .. . a=N;
```

- Similarly for a query such as the following, each predicate combined with `AND` uses approximately 125 bytes:

```
SELECT COUNT(*) FROM t
WHERE a=1 AND b=1 AND c=1 ... N;
```

- For a query with `IN()` predicates:

```
SELECT COUNT(*) FROM t
WHERE a IN (1,2, ..., M) AND b IN (1,2, ..., N);
```

  Each literal value in an `IN()` list counts as a predicate combined with `OR`. If there are two `IN()` lists, the number of predicates combined with `OR` is the product of the number of literal values in each list. Thus, the number of predicates combined with `OR` in the preceding case is *`M`* × *`N`*.

8.2.1.3 Index Merge Optimization

The *Index Merge* access method retrieves rows with multiple `range` scans and merges their results into one. This access method merges index scans from a single table only, not scans across multiple tables. The merge can produce unions, intersections, or unions-of-intersections of its underlying scans.

Example queries for which Index Merge may be used:

```
SELECT * FROM tbl_name WHERE key1 = 10 OR key2 = 20;

SELECT * FROM tbl_name
  WHERE (key1 = 10 OR key2 = 20) AND non_key = 30;

SELECT * FROM t1, t2
  WHERE (t1.key1 IN (1,2) OR t1.key2 LIKE 'value%')
  AND t2.key1 = t1.some_col;

SELECT * FROM t1, t2
  WHERE t1.key1 = 1
  AND (t2.key1 = t1.some_col OR t2.key2 = t1.some_col2);
```

Note

The Index Merge optimization algorithm has the following known limitations:

- If your query has a complex `WHERE` clause with deep `AND`/`OR` nesting and MySQL does not choose the optimal plan, try distributing terms using the following identity transformations:

```
(x AND y) OR z => (x OR z) AND (y OR z)
(x OR y) AND z => (x AND z) OR (y AND z)
```

- Index Merge is not applicable to full-text indexes.

In `EXPLAIN` output, the Index Merge method appears as `index_merge` in the `type` column. In this case, the `key` column contains a list of indexes used, and `key_len` contains a list of the longest key parts for those indexes.

The Index Merge access method has several algorithms, which are displayed in the `Extra` field of `EXPLAIN` output:

- `Using intersect(...)`

- `Using union(...)`
- `Using sort_union(...)`

The following sections describe these algorithms in greater detail. The optimizer chooses between different possible Index Merge algorithms and other access methods based on cost estimates of the various available options.

- Index Merge Intersection Access Algorithm
- Index Merge Union Access Algorithm
- Index Merge Sort-Union Access Algorithm
- Influencing Index Merge Optimization

Index Merge Intersection Access Algorithm

This access algorithm is applicable when a `WHERE` clause is converted to several range conditions on different keys combined with `AND`, and each condition is one of the following:

- An *`N`*-part expression of this form, where the index has exactly *`N`* parts (that is, all index parts are covered):

  ```
  key_part1 = const1 AND key_part2 = const2 ... AND key_partN = constN
  ```

- Any range condition over the primary key of an `InnoDB` table.

Examples:

```
SELECT * FROM innodb_table
  WHERE primary_key < 10 AND key_col1 = 20;

SELECT * FROM tbl_name
  WHERE key1_part1 = 1 AND key1_part2 = 2 AND key2 = 2;
```

The Index Merge intersection algorithm performs simultaneous scans on all used indexes and produces the intersection of row sequences that it receives from the merged index scans.

If all columns used in the query are covered by the used indexes, full table rows are not retrieved (`EXPLAIN` output contains `Using index` in `Extra` field in this case). Here is an example of such a query:

```
SELECT COUNT(*) FROM t1 WHERE key1 = 1 AND key2 = 1;
```

If the used indexes do not cover all columns used in the query, full rows are retrieved only when the range conditions for all used keys are satisfied.

If one of the merged conditions is a condition over the primary key of an `InnoDB` table, it is not used for row retrieval, but is used to filter out rows retrieved using other conditions.

Index Merge Union Access Algorithm

The criteria for this algorithm are similar to those for the Index Merge intersection algorithm. The algorithm is applicable when the table's `WHERE` clause is converted to several range conditions on different keys combined with `OR`, and each condition is one of the following:

- An *`N`*-part expression of this form, where the index has exactly *`N`* parts (that is, all index parts are covered):

  ```
  key_part1 = const1 OR key_part2 = const2 ... OR key_partN = constN
  ```

- Any range condition over a primary key of an `InnoDB` table.

- A condition for which the Index Merge intersection algorithm is applicable.

Examples:

```
SELECT * FROM t1
  WHERE key1 = 1 OR key2 = 2 OR key3 = 3;

SELECT * FROM innodb_table
  WHERE (key1 = 1 AND key2 = 2)
     OR (key3 = 'foo' AND key4 = 'bar') AND key5 = 5;
```

Index Merge Sort-Union Access Algorithm

This access algorithm is applicable when the `WHERE` clause is converted to several range conditions combined by `OR`, but the Index Merge union algorithm is not applicable.

Examples:

```
SELECT * FROM tbl_name
  WHERE key_col1 < 10 OR key_col2 < 20;

SELECT * FROM tbl_name
  WHERE (key_col1 > 10 OR key_col2 = 20) AND nonkey_col = 30;
```

The difference between the sort-union algorithm and the union algorithm is that the sort-union algorithm must first fetch row IDs for all rows and sort them before returning any rows.

Influencing Index Merge Optimization

Use of Index Merge is subject to the value of the `index_merge`, `index_merge_intersection`, `index_merge_union`, and `index_merge_sort_union` flags of the `optimizer_switch` system variable. See Section 8.9.2, “Switchable Optimizations”. By default, all those flags are `on`. To enable only certain algorithms, set `index_merge` to `off`, and enable only such of the others as should be permitted.

In addition to using the `optimizer_switch` system variable to control optimizer use of the Index Merge algorithms session-wide, MySQL supports optimizer hints to influence the optimizer on a per-statement basis. See Section 8.9.3, “Optimizer Hints”.

8.2.1.4 Hash Join Optimization

By default, MySQL (8.0.18 and later) employs hash joins whenever possible. It is possible to control whether hash joins are employed using one of the `BNL` and `NO_BNL` optimizer hints, or by setting `block_nested_loop=on` or `block_nested_loop=off` as part of the setting for the optimizer_switch server system variable.

Note

MySQL 8.0.18 supported setting a `hash_join` flag in `optimizer_switch`, as well as the optimizer hints `HASH_JOIN` and `NO_HASH_JOIN`. In MySQL 8.0.19 and later, none of these have any effect any longer.

Beginning with MySQL 8.0.18, MySQL employs a hash join for any query for which each join has an equi-join condition, and in which there are no indexes that can be applied to any join conditions, such as this one:

```
SELECT *
    FROM t1
    JOIN t2
        ON t1.c1=t2.c1;
```

A hash join can also be used when there are one or more indexes that can be used for single-table predicates.

A hash join is usually faster than and is intended to be used in such cases instead of the block nested loop algorithm (see Block Nested-Loop Join Algorithm) employed in previous versions of MySQL. Beginning with MySQL 8.0.20, support for block nested loop is removed, and the server employs a hash join wherever a block nested loop would have been used previously.

In the example just shown and the remaining examples in this section, we assume that the three tables `t1`, `t2`, and `t3` have been created using the following statements:

```
CREATE TABLE t1 (c1 INT, c2 INT);
CREATE TABLE t2 (c1 INT, c2 INT);
CREATE TABLE t3 (c1 INT, c2 INT);
```

You can see that a hash join is being employed by using `EXPLAIN`, like this:

```
mysql> EXPLAIN
    -> SELECT * FROM t1
    ->     JOIN t2 ON t1.c1=t2.c1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra: NULL
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: t2
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra: Using where; Using join buffer (hash join)
```

(Prior to MySQL 8.0.20, it was necessary to include the `FORMAT=TREE` option to see whether hash joins were being used for a given join.)

`EXPLAIN ANALYZE` also displays information about hash joins used.

The hash join is used for queries involving multiple joins as well, as long as at least one join condition for each pair of tables is an equi-join, like the query shown here:

```
SELECT * FROM t1
    JOIN t2 ON (t1.c1 = t2.c1 AND t1.c2 < t2.c2)
    JOIN t3 ON (t2.c1 = t3.c1);
```

In cases like the one just shown, which makes use of an inner join, any extra conditions which are not equi-joins are applied as filters after the join is executed. (For outer joins, such as left joins, semijoins, and antijoins, they are printed as part of the join.) This can be seen here in the output of `EXPLAIN`:

```
mysql> EXPLAIN FORMAT=TREE
    -> SELECT *
    ->     FROM t1
    ->     JOIN t2
    ->         ON (t1.c1 = t2.c1 AND t1.c2 < t2.c2)
    ->     JOIN t3
    ->         ON (t2.c1 = t3.c1)\G
*************************** 1. row ***************************
EXPLAIN: -> Inner hash join (t3.c1 = t1.c1)  (cost=1.05 rows=1)
```

```
    -> Table scan on t3  (cost=0.35 rows=1)
    -> Hash
        -> Filter: (t1.c2 < t2.c2)  (cost=0.70 rows=1)
            -> Inner hash join (t2.c1 = t1.c1)  (cost=0.70 rows=1)
                -> Table scan on t2  (cost=0.35 rows=1)
                -> Hash
                    -> Table scan on t1  (cost=0.35 rows=1)
```

As also can be seen from the output just shown, multiple hash joins can be (and are) used for joins having multiple equi-join conditions.

Prior to MySQL 8.0.20, a hash join could not be used if any pair of joined tables did not have at least one equi-join condition, and the slower block nested loop algorithm was employed. In MySQL 8.0.20 and later, the hash join is used in such cases, as shown here:

```
mysql> EXPLAIN FORMAT=TREE
    -> SELECT * FROM t1
    ->     JOIN t2 ON (t1.c1 = t2.c1)
    ->     JOIN t3 ON (t2.c1 < t3.c1)\G
*************************** 1. row ***************************
EXPLAIN: -> Filter: (t1.c1 < t3.c1)  (cost=1.05 rows=1)
    -> Inner hash join (no condition)  (cost=1.05 rows=1)
        -> Table scan on t3  (cost=0.35 rows=1)
        -> Hash
            -> Inner hash join (t2.c1 = t1.c1)  (cost=0.70 rows=1)
                -> Table scan on t2  (cost=0.35 rows=1)
                -> Hash
                    -> Table scan on t1  (cost=0.35 rows=1)
```

(Additional examples are provided later in this section.)

A hash join is also applied for a Cartesian product—that is, when no join condition is specified, as shown here:

```
mysql> EXPLAIN FORMAT=TREE
    -> SELECT *
    ->     FROM t1
    ->     JOIN t2
    ->     WHERE t1.c2 > 50\G
*************************** 1. row ***************************
EXPLAIN: -> Inner hash join  (cost=0.70 rows=1)
    -> Table scan on t2  (cost=0.35 rows=1)
    -> Hash
        -> Filter: (t1.c2 > 50)  (cost=0.35 rows=1)
            -> Table scan on t1  (cost=0.35 rows=1)
```

In MySQL 8.0.20 and later, it is no longer necessary for the join to contain at least one equi-join condition in order for a hash join to be used. This means that the types of queries which can be optimized using hash joins include those in the following list (with examples):

- *Inner non-equi-join*:

  ```
  mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 JOIN t2 ON t1.c1 < t2.c1\G
  *************************** 1. row ***************************
  EXPLAIN: -> Filter: (t1.c1 < t2.c1)  (cost=4.70 rows=12)
      -> Inner hash join (no condition)  (cost=4.70 rows=12)
          -> Table scan on t2  (cost=0.08 rows=6)
          -> Hash
              -> Table scan on t1  (cost=0.85 rows=6)
  ```

- *Semijoin*:

  ```
  mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1
      ->     WHERE t1.c1 IN (SELECT t2.c2 FROM t2)\G
  *************************** 1. row ***************************
  EXPLAIN: -> Hash semijoin (t2.c2 = t1.c1)  (cost=0.70 rows=1)
      -> Table scan on t1  (cost=0.35 rows=1)
      -> Hash
          -> Table scan on t2  (cost=0.35 rows=1)
  ```

- *Antijoin*:

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t2
    ->     WHERE NOT EXISTS (SELECT * FROM t1 WHERE t1.c1 = t2.c1)\G
*************************** 1. row ***************************
EXPLAIN: -> Hash antijoin (t1.c1 = t2.c1)  (cost=0.70 rows=1)
    -> Table scan on t2  (cost=0.35 rows=1)
    -> Hash
        -> Table scan on t1  (cost=0.35 rows=1)

1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1276
Message: Field or reference 't3.t2.c1' of SELECT #2 was resolved in SELECT #1
```

- *Left outer join*:

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 LEFT JOIN t2 ON t1.c1 = t2.c1\G
*************************** 1. row ***************************
EXPLAIN: -> Left hash join (t2.c1 = t1.c1)  (cost=0.70 rows=1)
    -> Table scan on t1  (cost=0.35 rows=1)
    -> Hash
        -> Table scan on t2  (cost=0.35 rows=1)
```

- *Right outer join* (observe that MySQL rewrites all right outer joins as left outer joins):

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 RIGHT JOIN t2 ON t1.c1 = t2.c1\G
*************************** 1. row ***************************
EXPLAIN: -> Left hash join (t1.c1 = t2.c1)  (cost=0.70 rows=1)
    -> Table scan on t2  (cost=0.35 rows=1)
    -> Hash
        -> Table scan on t1  (cost=0.35 rows=1)
```

By default, MySQL 8.0.18 and later employs hash joins whenever possible. It is possible to control whether hash joins are employed using one of the `BNL` and `NO_BNL` optimizer hints.

(MySQL 8.0.18 supported `hash_join=on` or `hash_join=off` as part of the setting for the `optimizer_switch` server system variable as well as the optimizer hints `HASH_JOIN` or `NO_HASH_JOIN`. In MySQL 8.0.19 and later, these no longer have any effect.)

Memory usage by hash joins can be controlled using the `join_buffer_size` system variable; a hash join cannot use more memory than this amount. When the memory required for a hash join exceeds the amount available, MySQL handles this by using files on disk. If this happens, you should be aware that the join may not succeed if a hash join cannot fit into memory and it creates more files than set for `open_files_limit`. To avoid such problems, make either of the following changes:

- Increase `join_buffer_size` so that the hash join does not spill over to disk.
- Increase `open_files_limit`.

Beginning with MySQL 8.0.18, join buffers for hash joins are allocated incrementally; thus, you can set `join_buffer_size` higher without small queries allocating very large amounts of RAM, but outer joins allocate the entire buffer. In MySQL 8.0.20 and later, hash joins are used for outer joins (including antijoins and semijoins) as well, so this is no longer an issue.

8.2.1.5 Engine Condition Pushdown Optimization

This optimization improves the efficiency of direct comparisons between a nonindexed column and a constant. In such cases, the condition is “pushed down” to the storage engine for evaluation. This optimization can be used only by the `NDB` storage engine.

For NDB Cluster, this optimization can eliminate the need to send nonmatching rows over the network between the cluster's data nodes and the MySQL server that issued the query, and can speed up

queries where it is used by a factor of 5 to 10 times over cases where condition pushdown could be but is not used.

Suppose that an NDB Cluster table is defined as follows:

```
CREATE TABLE t1 (
    a INT,
    b INT,
    KEY(a)
) ENGINE=NDB;
```

Engine condition pushdown can be used with queries such as the one shown here, which includes a comparison between a nonindexed column and a constant:

```
SELECT a, b FROM t1 WHERE b = 10;
```

The use of engine condition pushdown can be seen in the output of `EXPLAIN`:

```
mysql> EXPLAIN SELECT a, b FROM t1 WHERE b = 10\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 10
        Extra: Using where with pushed condition
```

However, engine condition pushdown *cannot* be used with the following query:

```
SELECT a,b FROM t1 WHERE a = 10;
```

Engine condition pushdown is not applicable here because an index exists on column `a`. (An index access method would be more efficient and so would be chosen in preference to condition pushdown.)

Engine condition pushdown may also be employed when an indexed column is compared with a constant using a `>` or `<` operator:

```
mysql> EXPLAIN SELECT a, b FROM t1 WHERE a < 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: range
possible_keys: a
          key: a
      key_len: 5
          ref: NULL
         rows: 2
        Extra: Using where with pushed condition
```

Other supported comparisons for engine condition pushdown include the following:

- *`column`* `[NOT] LIKE` *`pattern`*

  `pattern` must be a string literal containing the pattern to be matched; for syntax, see Section 12.8.1, “String Comparison Functions and Operators”.

- *`column`* `IS [NOT] NULL`

- *`column`* `IN (`*`value_list`*`)`

  Each item in the *`value_list`* must be a constant, literal value.

- *`column`* `BETWEEN` *`constant1`* `AND` *`constant2`*

constant1 and *constant2* must each be a constant, literal value.

In all of the cases in the preceding list, it is possible for the condition to be converted into the form of one or more direct comparisons between a column and a constant.

Engine condition pushdown is enabled by default. To disable it at server startup, set the `optimizer_switch` system variable's `engine_condition_pushdown` flag to `off`. For example, in a `my.cnf` file, use these lines:

```
[mysqld]
optimizer_switch=engine_condition_pushdown=off
```

At runtime, disable condition pushdown like this:

```
SET optimizer_switch='engine_condition_pushdown=off';
```

Limitations. Engine condition pushdown is subject to the following limitations:

- Engine condition pushdown is supported only by the `NDB` storage engine.
- Prior to NDB 8.0.18, columns could be compared with constants or expressions which evaluate to constant values only. In NDB 8.0.18 and later, columns can be compared with one another as long as they are of exactly the same type, including the same signedness, length, character set, precision, and scale, where these are applicable.
- Columns used in comparisons cannot be of any of the `BLOB` or `TEXT` types. This exclusion extends to `JSON`, `BIT`, and `ENUM` columns as well.
- A string value to be compared with a column must use the same collation as the column.
- Joins are not directly supported; conditions involving multiple tables are pushed separately where possible. Use extended `EXPLAIN` output to determine which conditions are actually pushed down. See Section 8.8.3, “Extended EXPLAIN Output Format”.

Previously, engine condition pushdown was limited to terms referring to column values from the same table to which the condition was being pushed. Beginning with NDB 8.0.16, column values from tables earlier in the query plan can also be referred to from pushed conditions. This reduces the number of rows which must be handled by the SQL node during join processing. Filtering can be also performed in parallel in the LDM threads, rather than in a single `mysqld` process. This has the potential to improve performance of queries by a significant margin.

Beginning with NDB 8.0.20, an outer join using a scan can be pushed if there are no unpushable conditions on any table used in the same join nest, or on any table in join nests above it on which it depends. This is also true for a semijoin, provided the optimization strategy employed is `firstMatch` (see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”).

Join algorithms cannot be combined with referring columns from previous tables in the following two situations:

1. When any of the referred previous tables are in a join buffer. In this case, each row retrieved from the scan-filtered table is matched against every row in the buffer. This means that there is no single specific row from which column values can be fetched from when generating the scan filter.
2. When the column originates from a child operation in a pushed join. This is because rows referenced from ancestor operations in the join have not yet been retrieved when the scan filter is generated.

Beginning with NDB 8.0.27, columns from ancestor tables in a join can be pushed down, provided that they meet the requirements listed previously. An example of such a query, using the table `t1` created previously, is shown here:

```
mysql> EXPLAIN
```

```
    ->   SELECT * FROM t1 AS x
    ->   LEFT JOIN t1 AS y
    ->   ON x.a=0 AND y.b>=3\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: x
   partitions: p0,p1
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
     filtered: 100.00
        Extra: NULL
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: y
   partitions: p0,p1
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
     filtered: 100.00
        Extra: Using where; Using pushed condition (`test`.`y`.`b` >= 3); Using join buffer (hash join)
2 rows in set, 2 warnings (0.00 sec)
```

8.2.1.6 Index Condition Pushdown Optimization

Index Condition Pushdown (ICP) is an optimization for the case where MySQL retrieves rows from a table using an index. Without ICP, the storage engine traverses the index to locate rows in the base table and returns them to the MySQL server which evaluates the `WHERE` condition for the rows. With ICP enabled, and if parts of the `WHERE` condition can be evaluated by using only columns from the index, the MySQL server pushes this part of the `WHERE` condition down to the storage engine. The storage engine then evaluates the pushed index condition by using the index entry and only if this is satisfied is the row read from the table. ICP can reduce the number of times the storage engine must access the base table and the number of times the MySQL server must access the storage engine.

Applicability of the Index Condition Pushdown optimization is subject to these conditions:

- ICP is used for the `range`, `ref`, `eq_ref`, and `ref_or_null` access methods when there is a need to access full table rows.
- ICP can be used for `InnoDB` and `MyISAM` tables, including partitioned `InnoDB` and `MyISAM` tables.
- For `InnoDB` tables, ICP is used only for secondary indexes. The goal of ICP is to reduce the number of full-row reads and thereby reduce I/O operations. For `InnoDB` clustered indexes, the complete record is already read into the `InnoDB` buffer. Using ICP in this case does not reduce I/O.
- ICP is not supported with secondary indexes created on virtual generated columns. `InnoDB` supports secondary indexes on virtual generated columns.
- Conditions that refer to subqueries cannot be pushed down.
- Conditions that refer to stored functions cannot be pushed down. Storage engines cannot invoke stored functions.
- Triggered conditions cannot be pushed down. (For information about triggered conditions, see Section 8.2.2.3, “Optimizing Subqueries with the EXISTS Strategy”.)
- (*MySQL 8.0.30 and later*:) Conditions cannot be pushed down to derived tables containing references to system variables.

To understand how this optimization works, first consider how an index scan proceeds when Index Condition Pushdown is not used:

1. Get the next row, first by reading the index tuple, and then by using the index tuple to locate and read the full table row.
2. Test the part of the `WHERE` condition that applies to this table. Accept or reject the row based on the test result.

Using Index Condition Pushdown, the scan proceeds like this instead:

1. Get the next row's index tuple (but not the full table row).
2. Test the part of the `WHERE` condition that applies to this table and can be checked using only index columns. If the condition is not satisfied, proceed to the index tuple for the next row.
3. If the condition is satisfied, use the index tuple to locate and read the full table row.
4. Test the remaining part of the `WHERE` condition that applies to this table. Accept or reject the row based on the test result.

`EXPLAIN` output shows `Using index condition` in the `Extra` column when Index Condition Pushdown is used. It does not show `Using index` because that does not apply when full table rows must be read.

Suppose that a table contains information about people and their addresses and that the table has an index defined as `INDEX (zipcode, lastname, firstname)`. If we know a person's `zipcode` value but are not sure about the last name, we can search like this:

```
SELECT * FROM people
  WHERE zipcode='95054'
  AND lastname LIKE '%etrunia%'
  AND address LIKE '%Main Street%';
```

MySQL can use the index to scan through people with `zipcode='95054'`. The second part (`lastname LIKE '%etrunia%'`) cannot be used to limit the number of rows that must be scanned, so without Index Condition Pushdown, this query must retrieve full table rows for all people who have `zipcode='95054'`.

With Index Condition Pushdown, MySQL checks the `lastname LIKE '%etrunia%'` part before reading the full table row. This avoids reading full rows corresponding to index tuples that match the `zipcode` condition but not the `lastname` condition.

Index Condition Pushdown is enabled by default. It can be controlled with the `optimizer_switch` system variable by setting the `index_condition_pushdown` flag:

```
SET optimizer_switch = 'index_condition_pushdown=off';
SET optimizer_switch = 'index_condition_pushdown=on';
```

See Section 8.9.2, “Switchable Optimizations”.

8.2.1.7 Nested-Loop Join Algorithms

MySQL executes joins between tables using a nested-loop algorithm or variations on it.

- Nested-Loop Join Algorithm
- Block Nested-Loop Join Algorithm

Nested-Loop Join Algorithm

A simple nested-loop join (NLJ) algorithm reads rows from the first table in a loop one at a time, passing each row to a nested loop that processes the next table in the join. This process is repeated as many times as there remain tables to be joined.

Assume that a join between three tables `t1`, `t2`, and `t3` is to be executed using the following join types:

```
Table   Join Type
t1      range
t2      ref
t3      ALL
```

If a simple NLJ algorithm is used, the join is processed like this:

```
for each row in t1 matching range {
  for each row in t2 matching reference key {
    for each row in t3 {
      if row satisfies join conditions, send to client
    }
  }
}
```

Because the NLJ algorithm passes rows one at a time from outer loops to inner loops, it typically reads tables processed in the inner loops many times.

Block Nested-Loop Join Algorithm

A Block Nested-Loop (BNL) join algorithm uses buffering of rows read in outer loops to reduce the number of times that tables in inner loops must be read. For example, if 10 rows are read into a buffer and the buffer is passed to the next inner loop, each row read in the inner loop can be compared against all 10 rows in the buffer. This reduces by an order of magnitude the number of times the inner table must be read.

Prior to MySQL 8.0.18, this algorithm was applied for equi-joins when no indexes could be used; in MySQL 8.0.18 and later, the hash join optimization is employed in such cases. Starting with MySQL 8.0.20, the block nested loop is no longer used by MySQL, and a hash join is employed for in all cases where the block nested loop was used previously. See Section 8.2.1.4, “Hash Join Optimization”.

MySQL join buffering has these characteristics:

- Join buffering can be used when the join is of type `ALL` or `index` (in other words, when no possible keys can be used, and a full scan is done, of either the data or index rows, respectively), or `range`. Use of buffering is also applicable to outer joins, as described in Section 8.2.1.12, “Block Nested-Loop and Batched Key Access Joins”.
- A join buffer is never allocated for the first nonconstant table, even if it would be of type `ALL` or `index`.
- Only columns of interest to a join are stored in its join buffer, not whole rows.
- The `join_buffer_size` system variable determines the size of each join buffer used to process a query.
- One buffer is allocated for each join that can be buffered, so a given query might be processed using multiple join buffers.
- A join buffer is allocated prior to executing the join and freed after the query is done.

For the example join described previously for the NLJ algorithm (without buffering), the join is done as follows using join buffering:

```
for each row in t1 matching range {
  for each row in t2 matching reference key {
    store used columns from t1, t2 in join buffer
    if buffer is full {
      for each row in t3 {
        for each t1, t2 combination in join buffer {
          if row satisfies join conditions, send to client
```

```
        }
      }
      empty join buffer
    }
  }
}

if buffer is not empty {
  for each row in t3 {
    for each t1, t2 combination in join buffer {
      if row satisfies join conditions, send to client
    }
  }
}
```

If *S* is the size of each stored `t1`, `t2` combination in the join buffer and *C* is the number of combinations in the buffer, the number of times table `t3` is scanned is:

```
(S * C)/join_buffer_size + 1
```

The number of `t3` scans decreases as the value of `join_buffer_size` increases, up to the point when `join_buffer_size` is large enough to hold all previous row combinations. At that point, no speed is gained by making it larger.

8.2.1.8 Nested Join Optimization

The syntax for expressing joins permits nested joins. The following discussion refers to the join syntax described in Section 13.2.13.2, “JOIN Clause”.

The syntax of *`table_factor`* is extended in comparison with the SQL Standard. The latter accepts only *`table_reference`*, not a list of them inside a pair of parentheses. This is a conservative extension if we consider each comma in a list of *`table_reference`* items as equivalent to an inner join. For example:

```
SELECT * FROM t1 LEFT JOIN (t2, t3, t4)
                 ON (t2.a=t1.a AND t3.b=t1.b AND t4.c=t1.c)
```

Is equivalent to:

```
SELECT * FROM t1 LEFT JOIN (t2 CROSS JOIN t3 CROSS JOIN t4)
                 ON (t2.a=t1.a AND t3.b=t1.b AND t4.c=t1.c)
```

In MySQL, `CROSS JOIN` is syntactically equivalent to `INNER JOIN`; they can replace each other. In standard SQL, they are not equivalent. `INNER JOIN` is used with an `ON` clause; `CROSS JOIN` is used otherwise.

In general, parentheses can be ignored in join expressions containing only inner join operations. Consider this join expression:

```
t1 LEFT JOIN (t2 LEFT JOIN t3 ON t2.b=t3.b OR t2.b IS NULL)
   ON t1.a=t2.a
```

After removing parentheses and grouping operations to the left, that join expression transforms into this expression:

```
(t1 LEFT JOIN t2 ON t1.a=t2.a) LEFT JOIN t3
    ON t2.b=t3.b OR t2.b IS NULL
```

Yet, the two expressions are not equivalent. To see this, suppose that the tables `t1`, `t2`, and `t3` have the following state:

- Table `t1` contains rows `(1)`, `(2)`
- Table `t2` contains row `(1,101)`

- Table `t3` contains row `(101)`

In this case, the first expression returns a result set including the rows `(1,1,101,101)`, `(2,NULL,NULL,NULL)`, whereas the second expression returns the rows `(1,1,101,101)`, `(2,NULL,NULL,101)`:

```
mysql> SELECT *
       FROM t1
            LEFT JOIN
            (t2 LEFT JOIN t3 ON t2.b=t3.b OR t2.b IS NULL)
            ON t1.a=t2.a;
+------+------+------+------+
| a    | a    | b    | b    |
+------+------+------+------+
|    1 |    1 |  101 |  101 |
|    2 | NULL | NULL | NULL |
+------+------+------+------+

mysql> SELECT *
       FROM (t1 LEFT JOIN t2 ON t1.a=t2.a)
            LEFT JOIN t3
            ON t2.b=t3.b OR t2.b IS NULL;
+------+------+------+------+
| a    | a    | b    | b    |
+------+------+------+------+
|    1 |    1 |  101 |  101 |
|    2 | NULL | NULL |  101 |
+------+------+------+------+
```

In the following example, an outer join operation is used together with an inner join operation:

```
t1 LEFT JOIN (t2, t3) ON t1.a=t2.a
```

That expression cannot be transformed into the following expression:

```
t1 LEFT JOIN t2 ON t1.a=t2.a, t3
```

For the given table states, the two expressions return different sets of rows:

```
mysql> SELECT *
       FROM t1 LEFT JOIN (t2, t3) ON t1.a=t2.a;
+------+------+------+------+
| a    | a    | b    | b    |
+------+------+------+------+
|    1 |    1 |  101 |  101 |
|    2 | NULL | NULL | NULL |
+------+------+------+------+

mysql> SELECT *
       FROM t1 LEFT JOIN t2 ON t1.a=t2.a, t3;
+------+------+------+------+
| a    | a    | b    | b    |
+------+------+------+------+
|    1 |    1 |  101 |  101 |
|    2 | NULL | NULL |  101 |
+------+------+------+------+
```

Therefore, if we omit parentheses in a join expression with outer join operators, we might change the result set for the original expression.

More exactly, we cannot ignore parentheses in the right operand of the left outer join operation and in the left operand of a right join operation. In other words, we cannot ignore parentheses for the inner table expressions of outer join operations. Parentheses for the other operand (operand for the outer table) can be ignored.

The following expression:

```
(t1,t2) LEFT JOIN t3 ON P(t2.b,t3.b)
```

Is equivalent to this expression for any tables `t1,t2,t3` and any condition `P` over attributes `t2.b` and `t3.b`:

```
t1, t2 LEFT JOIN t3 ON P(t2.b,t3.b)
```

Whenever the order of execution of join operations in a join expression (*`joined_table`*) is not from left to right, we talk about nested joins. Consider the following queries:

```
SELECT * FROM t1 LEFT JOIN (t2 LEFT JOIN t3 ON t2.b=t3.b) ON t1.a=t2.a
  WHERE t1.a > 1

SELECT * FROM t1 LEFT JOIN (t2, t3) ON t1.a=t2.a
  WHERE (t2.b=t3.b OR t2.b IS NULL) AND t1.a > 1
```

Those queries are considered to contain these nested joins:

```
t2 LEFT JOIN t3 ON t2.b=t3.b
t2, t3
```

In the first query, the nested join is formed with a left join operation. In the second query, it is formed with an inner join operation.

In the first query, the parentheses can be omitted: The grammatical structure of the join expression dictates the same order of execution for join operations. For the second query, the parentheses cannot be omitted, although the join expression here can be interpreted unambiguously without them. In our extended syntax, the parentheses in `(t2, t3)` of the second query are required, although theoretically the query could be parsed without them: We still would have unambiguous syntactical structure for the query because `LEFT JOIN` and `ON` play the role of the left and right delimiters for the expression `(t2,t3)`.

The preceding examples demonstrate these points:

- For join expressions involving only inner joins (and not outer joins), parentheses can be removed and joins evaluated left to right. In fact, tables can be evaluated in any order.
- The same is not true, in general, for outer joins or for outer joins mixed with inner joins. Removal of parentheses may change the result.

Queries with nested outer joins are executed in the same pipeline manner as queries with inner joins. More exactly, a variation of the nested-loop join algorithm is exploited. Recall the algorithm by which the nested-loop join executes a query (see Section 8.2.1.7, “Nested-Loop Join Algorithms”). Suppose that a join query over 3 tables `T1,T2,T3` has this form:

```
SELECT * FROM T1 INNER JOIN T2 ON P1(T1,T2)
                 INNER JOIN T3 ON P2(T2,T3)
  WHERE P(T1,T2,T3)
```

Here, `P1(T1,T2)` and `P2(T3,T3)` are some join conditions (on expressions), whereas `P(T1,T2,T3)` is a condition over columns of tables `T1,T2,T3`.

The nested-loop join algorithm would execute this query in the following manner:

```
FOR each row t1 in T1 {
  FOR each row t2 in T2 such that P1(t1,t2) {
    FOR each row t3 in T3 such that P2(t2,t3) {
      IF P(t1,t2,t3) {
         t:=t1||t2||t3; OUTPUT t;
      }
    }
  }
}
```

The notation `t1||t2||t3` indicates a row constructed by concatenating the columns of rows `t1`, `t2`, and `t3`. In some of the following examples, `NULL` where a table name appears means a row in which `NULL` is used for each column of that table. For example, `t1||t2||NULL` indicates a row constructed

by concatenating the columns of rows `t1` and `t2`, and `NULL` for each column of `t3`. Such a row is said to be `NULL`-complemented.

Now consider a query with nested outer joins:

```
SELECT * FROM T1 LEFT JOIN
             (T2 LEFT JOIN T3 ON P2(T2,T3))
             ON P1(T1,T2)
  WHERE P(T1,T2,T3)
```

For this query, modify the nested-loop pattern to obtain:

```
FOR each row t1 in T1 {
  BOOL f1:=FALSE;
  FOR each row t2 in T2 such that P1(t1,t2) {
    BOOL f2:=FALSE;
    FOR each row t3 in T3 such that P2(t2,t3) {
      IF P(t1,t2,t3) {
        t:=t1||t2||t3; OUTPUT t;
      }
      f2=TRUE;
      f1=TRUE;
    }
    IF (!f2) {
      IF P(t1,t2,NULL) {
        t:=t1||t2||NULL; OUTPUT t;
      }
      f1=TRUE;
    }
  }
  IF (!f1) {
    IF P(t1,NULL,NULL) {
      t:=t1||NULL||NULL; OUTPUT t;
    }
  }
}
```

In general, for any nested loop for the first inner table in an outer join operation, a flag is introduced that is turned off before the loop and is checked after the loop. The flag is turned on when for the current row from the outer table a match from the table representing the inner operand is found. If at the end of the loop cycle the flag is still off, no match has been found for the current row of the outer table. In this case, the row is complemented by `NULL` values for the columns of the inner tables. The result row is passed to the final check for the output or into the next nested loop, but only if the row satisfies the join condition of all embedded outer joins.

In the example, the outer join table expressed by the following expression is embedded:

```
(T2 LEFT JOIN T3 ON P2(T2,T3))
```

For the query with inner joins, the optimizer could choose a different order of nested loops, such as this one:

```
FOR each row t3 in T3 {
  FOR each row t2 in T2 such that P2(t2,t3) {
    FOR each row t1 in T1 such that P1(t1,t2) {
      IF P(t1,t2,t3) {
         t:=t1||t2||t3; OUTPUT t;
      }
    }
  }
}
```

For queries with outer joins, the optimizer can choose only such an order where loops for outer tables precede loops for inner tables. Thus, for our query with outer joins, only one nesting order is possible. For the following query, the optimizer evaluates two different nestings. In both nestings, `T1` must be processed in the outer loop because it is used in an outer join. `T2` and `T3` are used in an inner join, so that join must be processed in the inner loop. However, because the join is an inner join, `T2` and `T3` can be processed in either order.

```
SELECT * T1 LEFT JOIN (T2,T3) ON P1(T1,T2) AND P2(T1,T3)
  WHERE P(T1,T2,T3)
```

One nesting evaluates `T2`, then `T3`:

```
FOR each row t1 in T1 {
  BOOL f1:=FALSE;
  FOR each row t2 in T2 such that P1(t1,t2) {
    FOR each row t3 in T3 such that P2(t1,t3) {
      IF P(t1,t2,t3) {
        t:=t1||t2||t3; OUTPUT t;
      }
      f1:=TRUE
    }
  }
  IF (!f1) {
    IF P(t1,NULL,NULL) {
      t:=t1||NULL||NULL; OUTPUT t;
    }
  }
}
```

The other nesting evaluates `T3`, then `T2`:

```
FOR each row t1 in T1 {
  BOOL f1:=FALSE;
  FOR each row t3 in T3 such that P2(t1,t3) {
    FOR each row t2 in T2 such that P1(t1,t2) {
      IF P(t1,t2,t3) {
        t:=t1||t2||t3; OUTPUT t;
      }
      f1:=TRUE
    }
  }
  IF (!f1) {
    IF P(t1,NULL,NULL) {
      t:=t1||NULL||NULL; OUTPUT t;
    }
  }
}
```

When discussing the nested-loop algorithm for inner joins, we omitted some details whose impact on the performance of query execution may be huge. We did not mention so-called “pushed-down” conditions. Suppose that our `WHERE` condition `P(T1,T2,T3)` can be represented by a conjunctive formula:

```
P(T1,T2,T2) = C1(T1) AND C2(T2) AND C3(T3).
```

In this case, MySQL actually uses the following nested-loop algorithm for the execution of the query with inner joins:

```
FOR each row t1 in T1 such that C1(t1) {
  FOR each row t2 in T2 such that P1(t1,t2) AND C2(t2)  {
    FOR each row t3 in T3 such that P2(t2,t3) AND C3(t3) {
      IF P(t1,t2,t3) {
         t:=t1||t2||t3; OUTPUT t;
      }
    }
  }
}
```

You see that each of the conjuncts `C1(T1)`, `C2(T2)`, `C3(T3)` are pushed out of the most inner loop to the most outer loop where it can be evaluated. If `C1(T1)` is a very restrictive condition, this condition pushdown may greatly reduce the number of rows from table `T1` passed to the inner loops. As a result, the execution time for the query may improve immensely.

For a query with outer joins, the `WHERE` condition is to be checked only after it has been found that the current row from the outer table has a match in the inner tables. Thus, the optimization of pushing

conditions out of the inner nested loops cannot be applied directly to queries with outer joins. Here we must introduce conditional pushed-down predicates guarded by the flags that are turned on when a match has been encountered.

Recall this example with outer joins:

```
P(T1,T2,T3)=C1(T1) AND C(T2) AND C3(T3)
```

For that example, the nested-loop algorithm using guarded pushed-down conditions looks like this:

```
FOR each row t1 in T1 such that C1(t1) {
  BOOL f1:=FALSE;
  FOR each row t2 in T2
      such that P1(t1,t2) AND (f1?C2(t2):TRUE) {
    BOOL f2:=FALSE;
    FOR each row t3 in T3
        such that P2(t2,t3) AND (f1&&f2?C3(t3):TRUE) {
      IF (f1&&f2?TRUE:(C2(t2) AND C3(t3))) {
        t:=t1||t2||t3; OUTPUT t;
      }
      f2=TRUE;
      f1=TRUE;
    }
    IF (!f2) {
      IF (f1?TRUE:C2(t2) && P(t1,t2,NULL)) {
        t:=t1||t2||NULL; OUTPUT t;
      }
      f1=TRUE;
    }
  }
  IF (!f1 && P(t1,NULL,NULL)) {
      t:=t1||NULL||NULL; OUTPUT t;
  }
}
```

In general, pushed-down predicates can be extracted from join conditions such as `P1(T1,T2)` and `P(T2,T3)`. In this case, a pushed-down predicate is guarded also by a flag that prevents checking the predicate for the `NULL`-complemented row generated by the corresponding outer join operation.

Access by key from one inner table to another in the same nested join is prohibited if it is induced by a predicate from the `WHERE` condition.

8.2.1.9 Outer Join Optimization

Outer joins include `LEFT JOIN` and `RIGHT JOIN`.

MySQL implements an *`A LEFT JOIN B join_specification`* as follows:

- Table *`B`* is set to depend on table *`A`* and all tables on which *`A`* depends.
- Table *`A`* is set to depend on all tables (except *`B`*) that are used in the `LEFT JOIN` condition.
- The `LEFT JOIN` condition is used to decide how to retrieve rows from table *`B`*. (In other words, any condition in the `WHERE` clause is not used.)
- All standard join optimizations are performed, with the exception that a table is always read after all tables on which it depends. If there is a circular dependency, an error occurs.
- All standard `WHERE` optimizations are performed.
- If there is a row in *`A`* that matches the `WHERE` clause, but there is no row in *`B`* that matches the `ON` condition, an extra *`B`* row is generated with all columns set to `NULL`.
- If you use `LEFT JOIN` to find rows that do not exist in some table and you have the following test: *`col_name`* `IS NULL` in the `WHERE` part, where *`col_name`* is a column that is declared as `NOT`

`NULL`, MySQL stops searching for more rows (for a particular key combination) after it has found one row that matches the `LEFT JOIN` condition.

The `RIGHT JOIN` implementation is analogous to that of `LEFT JOIN` with the table roles reversed. Right joins are converted to equivalent left joins, as described in Section 8.2.1.10, “Outer Join Simplification”.

For a `LEFT JOIN`, if the `WHERE` condition is always false for the generated `NULL` row, the `LEFT JOIN` is changed to an inner join. For example, the `WHERE` clause would be false in the following query if `t2.column1` were `NULL`:

```
SELECT * FROM t1 LEFT JOIN t2 ON (column1) WHERE t2.column2=5;
```

Therefore, it is safe to convert the query to an inner join:

```
SELECT * FROM t1, t2 WHERE t2.column2=5 AND t1.column1=t2.column1;
```

In MySQL 8.0.14 and later, trivial `WHERE` conditions arising from constant literal expressions are removed during preparation, rather than at a later stage in optimization, by which time joins have already been simplified. Earlier removal of trivial conditions allows the optimizer to convert outer joins to inner joins; this can result in improved plans for queries with outer joins containing trivial conditions in the `WHERE` clause, such as this one:

```
SELECT * FROM t1 LEFT JOIN t2 ON condition_1 WHERE condition_2 OR 0 = 1
```

The optimizer now sees during preparation that 0 = 1 is always false, making `OR 0 = 1` redundant, and removes it, leaving this:

```
SELECT * FROM t1 LEFT JOIN t2 ON condition_1 where condition_2
```

Now the optimizer can rewrite the query as an inner join, like this:

```
SELECT * FROM t1 JOIN t2 WHERE condition_1 AND condition_2
```

Now the optimizer can use table `t2` before table `t1` if doing so would result in a better query plan. To provide a hint about the table join order, use optimizer hints; see Section 8.9.3, “Optimizer Hints”. Alternatively, use `STRAIGHT_JOIN`; see Section 13.2.13, “SELECT Statement”. However, `STRAIGHT_JOIN` may prevent indexes from being used because it disables semijoin transformations; see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”.

8.2.1.10 Outer Join Simplification

Table expressions in the `FROM` clause of a query are simplified in many cases.

At the parser stage, queries with right outer join operations are converted to equivalent queries containing only left join operations. In the general case, the conversion is performed such that this right join:

```
(T1, ...) RIGHT JOIN (T2, ...) ON P(T1, ..., T2, ...)
```

Becomes this equivalent left join:

```
(T2, ...) LEFT JOIN (T1, ...) ON P(T1, ..., T2, ...)
```

All inner join expressions of the form `T1 INNER JOIN T2 ON P(T1,T2)` are replaced by the list `T1,T2`, `P(T1,T2)` being joined as a conjunct to the `WHERE` condition (or to the join condition of the embedding join, if there is any).

When the optimizer evaluates plans for outer join operations, it takes into consideration only plans where, for each such operation, the outer tables are accessed before the inner tables. The optimizer

choices are limited because only such plans enable outer joins to be executed using the nested-loop algorithm.

Consider a query of this form, where `R(T2)` greatly narrows the number of matching rows from table `T2`:

```
SELECT * T1 FROM T1
  LEFT JOIN T2 ON P1(T1,T2)
  WHERE P(T1,T2) AND R(T2)
```

If the query is executed as written, the optimizer has no choice but to access the less-restricted table `T1` before the more-restricted table `T2`, which may produce a very inefficient execution plan.

Instead, MySQL converts the query to a query with no outer join operation if the `WHERE` condition is null-rejected. (That is, it converts the outer join to an inner join.) A condition is said to be null-rejected for an outer join operation if it evaluates to `FALSE` or `UNKNOWN` for any `NULL`-complemented row generated for the operation.

Thus, for this outer join:

```
T1 LEFT JOIN T2 ON T1.A=T2.A
```

Conditions such as these are null-rejected because they cannot be true for any `NULL`-complemented row (with `T2` columns set to `NULL`):

```
T2.B IS NOT NULL
T2.B > 3
T2.C <= T1.C
T2.B < 2 OR T2.C > 1
```

Conditions such as these are not null-rejected because they might be true for a `NULL`-complemented row:

```
T2.B IS NULL
T1.B < 3 OR T2.B IS NOT NULL
T1.B < 3 OR T2.B > 3
```

The general rules for checking whether a condition is null-rejected for an outer join operation are simple:

- It is of the form `A IS NOT NULL`, where `A` is an attribute of any of the inner tables
- It is a predicate containing a reference to an inner table that evaluates to `UNKNOWN` when one of its arguments is `NULL`
- It is a conjunction containing a null-rejected condition as a conjunct
- It is a disjunction of null-rejected conditions

A condition can be null-rejected for one outer join operation in a query and not null-rejected for another. In this query, the `WHERE` condition is null-rejected for the second outer join operation but is not null-rejected for the first one:

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
                 LEFT JOIN T3 ON T3.B=T1.B
  WHERE T3.C > 0
```

If the `WHERE` condition is null-rejected for an outer join operation in a query, the outer join operation is replaced by an inner join operation.

For example, in the preceding query, the second outer join is null-rejected and can be replaced by an inner join:

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
                 INNER JOIN T3 ON T3.B=T1.B
```

```
  WHERE T3.C > 0
```

For the original query, the optimizer evaluates only plans compatible with the single table-access order `T1,T2,T3`. For the rewritten query, it additionally considers the access order `T3,T1,T2`.

A conversion of one outer join operation may trigger a conversion of another. Thus, the query:

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
                 LEFT JOIN T3 ON T3.B=T2.B
  WHERE T3.C > 0
```

Is first converted to the query:

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
                 INNER JOIN T3 ON T3.B=T2.B
  WHERE T3.C > 0
```

Which is equivalent to the query:

```
SELECT * FROM (T1 LEFT JOIN T2 ON T2.A=T1.A), T3
  WHERE T3.C > 0 AND T3.B=T2.B
```

The remaining outer join operation can also be replaced by an inner join because the condition `T3.B=T2.B` is null-rejected. This results in a query with no outer joins at all:

```
SELECT * FROM (T1 INNER JOIN T2 ON T2.A=T1.A), T3
  WHERE T3.C > 0 AND T3.B=T2.B
```

Sometimes the optimizer succeeds in replacing an embedded outer join operation, but cannot convert the embedding outer join. The following query:

```
SELECT * FROM T1 LEFT JOIN
              (T2 LEFT JOIN T3 ON T3.B=T2.B)
              ON T2.A=T1.A
  WHERE T3.C > 0
```

Is converted to:

```
SELECT * FROM T1 LEFT JOIN
              (T2 INNER JOIN T3 ON T3.B=T2.B)
              ON T2.A=T1.A
  WHERE T3.C > 0
```

That can be rewritten only to the form still containing the embedding outer join operation:

```
SELECT * FROM T1 LEFT JOIN
              (T2,T3)
              ON (T2.A=T1.A AND T3.B=T2.B)
  WHERE T3.C > 0
```

Any attempt to convert an embedded outer join operation in a query must take into account the join condition for the embedding outer join together with the `WHERE` condition. In this query, the `WHERE` condition is not null-rejected for the embedded outer join, but the join condition of the embedding outer join `T2.A=T1.A AND T3.C=T1.C` is null-rejected:

```
SELECT * FROM T1 LEFT JOIN
              (T2 LEFT JOIN T3 ON T3.B=T2.B)
              ON T2.A=T1.A AND T3.C=T1.C
  WHERE T3.D > 0 OR T1.D > 0
```

Consequently, the query can be converted to:

```
SELECT * FROM T1 LEFT JOIN
              (T2, T3)
              ON T2.A=T1.A AND T3.C=T1.C AND T3.B=T2.B
  WHERE T3.D > 0 OR T1.D > 0
```

8.2.1.11 Multi-Range Read Optimization

Reading rows using a range scan on a secondary index can result in many random disk accesses to the base table when the table is large and not stored in the storage engine's cache. With the Disk-Sweep Multi-Range Read (MRR) optimization, MySQL tries to reduce the number of random disk access for range scans by first scanning the index only and collecting the keys for the relevant rows. Then the keys are sorted and finally the rows are retrieved from the base table using the order of the primary key. The motivation for Disk-sweep MRR is to reduce the number of random disk accesses and instead achieve a more sequential scan of the base table data.

The Multi-Range Read optimization provides these benefits:

- MRR enables data rows to be accessed sequentially rather than in random order, based on index tuples. The server obtains a set of index tuples that satisfy the query conditions, sorts them according to data row ID order, and uses the sorted tuples to retrieve data rows in order. This makes data access more efficient and less expensive.
- MRR enables batch processing of requests for key access for operations that require access to data rows through index tuples, such as range index scans and equi-joins that use an index for the join attribute. MRR iterates over a sequence of index ranges to obtain qualifying index tuples. As these results accumulate, they are used to access the corresponding data rows. It is not necessary to acquire all index tuples before starting to read data rows.

The MRR optimization is not supported with secondary indexes created on virtual generated columns. `InnoDB` supports secondary indexes on virtual generated columns.

The following scenarios illustrate when MRR optimization can be advantageous:

Scenario A: MRR can be used for `InnoDB` and `MyISAM` tables for index range scans and equi-join operations.

1. A portion of the index tuples are accumulated in a buffer.
2. The tuples in the buffer are sorted by their data row ID.
3. Data rows are accessed according to the sorted index tuple sequence.

Scenario B: MRR can be used for `NDB` tables for multiple-range index scans or when performing an equi-join by an attribute.

1. A portion of ranges, possibly single-key ranges, is accumulated in a buffer on the central node where the query is submitted.
2. The ranges are sent to the execution nodes that access data rows.
3. The accessed rows are packed into packages and sent back to the central node.
4. The received packages with data rows are placed in a buffer.
5. Data rows are read from the buffer.

When MRR is used, the `Extra` column in `EXPLAIN` output shows `Using MRR`.

`InnoDB` and `MyISAM` do not use MRR if full table rows need not be accessed to produce the query result. This is the case if results can be produced entirely on the basis on information in the index tuples (through a covering index); MRR provides no benefit.

Two `optimizer_switch` system variable flags provide an interface to the use of MRR optimization. The `mrr` flag controls whether MRR is enabled. If `mrr` is enabled (`on`), the `mrr_cost_based` flag controls whether the optimizer attempts to make a cost-based choice between using and not using MRR (`on`) or uses MRR whenever possible (`off`). By default, `mrr` is `on` and `mrr_cost_based` is `on`. See Section 8.9.2, “Switchable Optimizations”.

For MRR, a storage engine uses the value of the `read_rnd_buffer_size` system variable as a guideline for how much memory it can allocate for its buffer. The engine uses up to `read_rnd_buffer_size` bytes and determines the number of ranges to process in a single pass.

8.2.1.12 Block Nested-Loop and Batched Key Access Joins

In MySQL, a Batched Key Access (BKA) Join algorithm is available that uses both index access to the joined table and a join buffer. The BKA algorithm supports inner join, outer join, and semijoin operations, including nested outer joins. Benefits of BKA include improved join performance due to more efficient table scanning. Also, the Block Nested-Loop (BNL) Join algorithm previously used only for inner joins is extended and can be employed for outer join and semijoin operations, including nested outer joins.

The following sections discuss the join buffer management that underlies the extension of the original BNL algorithm, the extended BNL algorithm, and the BKA algorithm. For information about semijoin strategies, see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”

- Join Buffer Management for Block Nested-Loop and Batched Key Access Algorithms
- Block Nested-Loop Algorithm for Outer Joins and Semijoins
- Batched Key Access Joins
- Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms

Join Buffer Management for Block Nested-Loop and Batched Key Access Algorithms

MySQL can employ join buffers to execute not only inner joins without index access to the inner table, but also outer joins and semijoins that appear after subquery flattening. Moreover, a join buffer can be effectively used when there is an index access to the inner table.

The join buffer management code slightly more efficiently utilizes join buffer space when storing the values of the interesting row columns: No additional bytes are allocated in buffers for a row column if its value is `NULL`, and the minimum number of bytes is allocated for any value of the `VARCHAR` type.

The code supports two types of buffers, regular and incremental. Suppose that join buffer `B1` is employed to join tables `t1` and `t2` and the result of this operation is joined with table `t3` using join buffer `B2`:

- A regular join buffer contains columns from each join operand. If `B2` is a regular join buffer, each row *`r`* put into `B2` is composed of the columns of a row *`r1`* from `B1` and the interesting columns of a matching row *`r2`* from table `t3`.
- An incremental join buffer contains only columns from rows of the table produced by the second join operand. That is, it is incremental to a row from the first operand buffer. If `B2` is an incremental join buffer, it contains the interesting columns of the row *`r2`* together with a link to the row *`r1`* from `B1`.

Incremental join buffers are always incremental relative to a join buffer from an earlier join operation, so the buffer from the first join operation is always a regular buffer. In the example just given, the buffer `B1` used to join tables `t1` and `t2` must be a regular buffer.

Each row of the incremental buffer used for a join operation contains only the interesting columns of a row from the table to be joined. These columns are augmented with a reference to the interesting columns of the matched row from the table produced by the first join operand. Several rows in the incremental buffer can refer to the same row *`r`* whose columns are stored in the previous join buffers insofar as all these rows match row *`r`*.

Incremental buffers enable less frequent copying of columns from buffers used for previous join operations. This provides a savings in buffer space because in the general case a row produced by the first join operand can be matched by several rows produced by the second join operand. It is

unnecessary to make several copies of a row from the first operand. Incremental buffers also provide a savings in processing time due to the reduction in copying time.

In MySQL 8.0, the `block_nested_loop` flag of the `optimizer_switch` system variable works as follows:

- Prior to MySQL 8.0.20, it controls how the optimizer uses the Block Nested Loop join algorithm.
- In MySQL 8.0.18 and later, it also controls the use of hash joins (see Section 8.2.1.4, “Hash Join Optimization”).
- Beginning with MySQL 8.0.20, the flag controls hash joins only, and the block nested loop algorithm is no longer supported.

The `batched_key_access` flag controls how the optimizer uses the Batched Key Access join algorithms.

By default, `block_nested_loop` is `on` and `batched_key_access` is `off`. See Section 8.9.2, “Switchable Optimizations”. Optimizer hints may also be applied; see Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms.

For information about semijoin strategies, see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”

Block Nested-Loop Algorithm for Outer Joins and Semijoins

The original implementation of the MySQL BNL algorithm was extended to support outer join and semijoin operations (and was later superseded by the hash join algorithm; see Section 8.2.1.4, “Hash Join Optimization”).

When these operations are executed with a join buffer, each row put into the buffer is supplied with a match flag.

If an outer join operation is executed using a join buffer, each row of the table produced by the second operand is checked for a match against each row in the join buffer. When a match is found, a new extended row is formed (the original row plus columns from the second operand) and sent for further extensions by the remaining join operations. In addition, the match flag of the matched row in the buffer is enabled. After all rows of the table to be joined have been examined, the join buffer is scanned. Each row from the buffer that does not have its match flag enabled is extended by `NULL` complements (`NULL` values for each column in the second operand) and sent for further extensions by the remaining join operations.

In MySQL 8.0, the `block_nested_loop` flag of the `optimizer_switch` system variable works as follows:

- Prior to MySQL 8.0.20, it controls how the optimizer uses the Block Nested Loop join algorithm.
- In MySQL 8.0.18 and later, it also controls the use of hash joins (see Section 8.2.1.4, “Hash Join Optimization”).
- Beginning with MySQL 8.0.20, the flag controls hash joins only, and the block nested loop algorithm is no longer supported.

See Section 8.9.2, “Switchable Optimizations”, for more information. Optimizer hints may also be applied; see Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms.

In `EXPLAIN` output, use of BNL for a table is signified when the `Extra` value contains `Using join buffer (Block Nested Loop)` and the `type` value is `ALL`, `index`, or `range`.

For information about semijoin strategies, see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”

Batched Key Access Joins

MySQL implements a method of joining tables called the Batched Key Access (BKA) join algorithm. BKA can be applied when there is an index access to the table produced by the second join operand. Like the BNL join algorithm, the BKA join algorithm employs a join buffer to accumulate the interesting columns of the rows produced by the first operand of the join operation. Then the BKA algorithm builds keys to access the table to be joined for all rows in the buffer and submits these keys in a batch to the database engine for index lookups. The keys are submitted to the engine through the Multi-Range Read (MRR) interface (see Section 8.2.1.11, “Multi-Range Read Optimization”). After submission of the keys, the MRR engine functions perform lookups in the index in an optimal way, fetching the rows of the joined table found by these keys, and starts feeding the BKA join algorithm with matching rows. Each matching row is coupled with a reference to a row in the join buffer.

When BKA is used, the value of `join_buffer_size` defines how large the batch of keys is in each request to the storage engine. The larger the buffer, the more sequential access is made to the right hand table of a join operation, which can significantly improve performance.

For BKA to be used, the `batched_key_access` flag of the `optimizer_switch` system variable must be set to `on`. BKA uses MRR, so the `mrr` flag must also be `on`. Currently, the cost estimation for MRR is too pessimistic. Hence, it is also necessary for `mrr_cost_based` to be `off` for BKA to be used. The following setting enables BKA:

```
mysql> SET optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on';
```

There are two scenarios by which MRR functions execute:

- The first scenario is used for conventional disk-based storage engines such as `InnoDB` and `MyISAM`. For these engines, usually the keys for all rows from the join buffer are submitted to the MRR interface at once. Engine-specific MRR functions perform index lookups for the submitted keys, get row IDs (or primary keys) from them, and then fetch rows for all these selected row IDs one by one by request from BKA algorithm. Every row is returned with an association reference that enables access to the matched row in the join buffer. The rows are fetched by the MRR functions in an optimal way: They are fetched in the row ID (primary key) order. This improves performance because reads are in disk order rather than random order.

- The second scenario is used for remote storage engines such as `NDB`. A package of keys for a portion of rows from the join buffer, together with their associations, is sent by a MySQL Server (SQL node) to MySQL Cluster data nodes. In return, the SQL node receives a package (or several packages) of matching rows coupled with corresponding associations. The BKA join algorithm takes these rows and builds new joined rows. Then a new set of keys is sent to the data nodes and the rows from the returned packages are used to build new joined rows. The process continues until the last keys from the join buffer are sent to the data nodes, and the SQL node has received and joined all rows matching these keys. This improves performance because fewer key-bearing packages sent by the SQL node to the data nodes means fewer round trips between it and the data nodes to perform the join operation.

With the first scenario, a portion of the join buffer is reserved to store row IDs (primary keys) selected by index lookups and passed as a parameter to the MRR functions.

There is no special buffer to store keys built for rows from the join buffer. Instead, a function that builds the key for the next row in the buffer is passed as a parameter to the MRR functions.

In `EXPLAIN` output, use of BKA for a table is signified when the `Extra` value contains `Using join buffer (Batched Key Access)` and the `type` value is `ref` or `eq_ref`.

Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms

In addition to using the `optimizer_switch` system variable to control optimizer use of the BNL and BKA algorithms session-wide, MySQL supports optimizer hints to influence the optimizer on a per-statement basis. See Section 8.9.3, “Optimizer Hints”.

To use a BNL or BKA hint to enable join buffering for any inner table of an outer join, join buffering must be enabled for all inner tables of the outer join.

8.2.1.13 Condition Filtering

In join processing, prefix rows are those rows passed from one table in a join to the next. In general, the optimizer attempts to put tables with low prefix counts early in the join order to keep the number of row combinations from increasing rapidly. To the extent that the optimizer can use information about conditions on rows selected from one table and passed to the next, the more accurately it can compute row estimates and choose the best execution plan.

Without condition filtering, the prefix row count for a table is based on the estimated number of rows selected by the `WHERE` clause according to whichever access method the optimizer chooses. Condition filtering enables the optimizer to use other relevant conditions in the `WHERE` clause not taken into account by the access method, and thus improve its prefix row count estimates. For example, even though there might be an index-based access method that can be used to select rows from the current table in a join, there might also be additional conditions for the table in the `WHERE` clause that can filter (further restrict) the estimate for qualifying rows passed to the next table.

A condition contributes to the filtering estimate only if:

- It refers to the current table.
- It depends on a constant value or values from earlier tables in the join sequence.
- It was not already taken into account by the access method.

In `EXPLAIN` output, the `rows` column indicates the row estimate for the chosen access method, and the `filtered` column reflects the effect of condition filtering. `filtered` values are expressed as percentages. The maximum value is 100, which means no filtering of rows occurred. Values decreasing from 100 indicate increasing amounts of filtering.

The prefix row count (the number of rows estimated to be passed from the current table in a join to the next) is the product of the `rows` and `filtered` values. That is, the prefix row count is the estimated row count, reduced by the estimated filtering effect. For example, if `rows` is 1000 and `filtered` is 20%, condition filtering reduces the estimated row count of 1000 to a prefix row count of 1000 × 20% = 1000 × .2 = 200.

Consider the following query:

```
SELECT *
  FROM employee JOIN department ON employee.dept_no = department.dept_no
  WHERE employee.first_name = 'John'
  AND employee.hire_date BETWEEN '2018-01-01' AND '2018-06-01';
```

Suppose that the data set has these characteristics:

- The `employee` table has 1024 rows.
- The `department` table has 12 rows.
- Both tables have an index on `dept_no`.
- The `employee` table has an index on `first_name`.
- 8 rows satisfy this condition on `employee.first_name`:

  ```
  employee.first_name = 'John'
  ```

- 150 rows satisfy this condition on `employee.hire_date`:

  ```
  employee.hire_date BETWEEN '2018-01-01' AND '2018-06-01'
  ```

- 1 row satisfies both conditions:

  ```
  employee.first_name = 'John'
  ```

```
AND employee.hire_date BETWEEN '2018-01-01' AND '2018-06-01'
```

Without condition filtering, `EXPLAIN` produces output like this:

```
+----+------------+--------+-----------------+---------+---------+------+----------+
| id | table      | type   | possible_keys   | key     | ref     | rows | filtered |
+----+------------+--------+-----------------+---------+---------+------+----------+
| 1  | employee   | ref    | name,h_date,dept | name    | const   | 8    | 100.00   |
| 1  | department | eq_ref | PRIMARY         | PRIMARY | dept_no | 1    | 100.00   |
+----+------------+--------+-----------------+---------+---------+------+----------+
```

For `employee`, the access method on the `name` index picks up the 8 rows that match a name of `'John'`. No filtering is done (`filtered` is 100%), so all rows are prefix rows for the next table: The prefix row count is `rows` × `filtered` = 8 × 100% = 8.

With condition filtering, the optimizer additionally takes into account conditions from the `WHERE` clause not taken into account by the access method. In this case, the optimizer uses heuristics to estimate a filtering effect of 16.31% for the `BETWEEN` condition on `employee.hire_date`. As a result, `EXPLAIN` produces output like this:

```
+----+------------+--------+-----------------+---------+---------+------+----------+
| id | table      | type   | possible_keys   | key     | ref     | rows | filtered |
+----+------------+--------+-----------------+---------+---------+------+----------+
| 1  | employee   | ref    | name,h_date,dept | name    | const   | 8    | 16.31    |
| 1  | department | eq_ref | PRIMARY         | PRIMARY | dept_no | 1    | 100.00   |
+----+------------+--------+-----------------+---------+---------+------+----------+
```

Now the prefix row count is `rows` × `filtered` = 8 × 16.31% = 1.3, which more closely reflects actual data set.

Normally, the optimizer does not calculate the condition filtering effect (prefix row count reduction) for the last joined table because there is no next table to pass rows to. An exception occurs for `EXPLAIN`: To provide more information, the filtering effect is calculated for all joined tables, including the last one.

To control whether the optimizer considers additional filtering conditions, use the `condition_fanout_filter` flag of the `optimizer_switch` system variable (see Section 8.9.2, “Switchable Optimizations”). This flag is enabled by default but can be disabled to suppress condition filtering (for example, if a particular query is found to yield better performance without it).

If the optimizer overestimates the effect of condition filtering, performance may be worse than if condition filtering is not used. In such cases, these techniques may help:

- If a column is not indexed, index it so that the optimizer has some information about the distribution of column values and can improve its row estimates.
- Similarly, if no column histogram information is available, generate a histogram (see Section 8.9.6, “Optimizer Statistics”).
- Change the join order. Ways to accomplish this include join-order optimizer hints (see Section 8.9.3, “Optimizer Hints”), `STRAIGHT_JOIN` immediately following the `SELECT`, and the `STRAIGHT_JOIN` join operator.
- Disable condition filtering for the session:

  ```
  SET optimizer_switch = 'condition_fanout_filter=off';
  ```

  Or, for a given query, using an optimizer hint:

  ```
  SELECT /*+ SET_VAR(optimizer_switch = 'condition_fanout_filter=off') */ ...
  ```

8.2.1.14 Constant-Folding Optimization

Comparisons between constants and column values in which the constant value is out of range or of the wrong type with respect to the column type are now handled once during query optimization rather

row-by-row than during execution. The comparisons that can be treated in this manner are `>`, `>=`, `<`, `<=`, `<>`/`!=`, `=`, and `<=>`.

Consider the table created by the following statement:

```
CREATE TABLE t (c TINYINT UNSIGNED NOT NULL);
```

The `WHERE` condition in the query `SELECT * FROM t WHERE c < 256` contains the integral constant 256 which is out of range for a `TINYINT UNSIGNED` column. Previously, this was handled by treating both operands as the larger type, but now, since any allowed value for `c` is less than the constant, the `WHERE` expression can instead be folded as `WHERE 1`, so that the query is rewritten as `SELECT * FROM t WHERE 1`.

This makes it possible for the optimizer to remove the `WHERE` expression altogether. If the column `c` were nullable (that is, defined only as `TINYINT UNSIGNED`) the query would be rewritten like this:

```
SELECT * FROM t WHERE ti IS NOT NULL
```

Folding is performed for constants compared to supported MySQL column types as follows:

- **Integer column type.** Integer types are compared with constants of the following types as described here:
  - **Integer value.** If the constant is out of range for the column type, the comparison is folded to `1` or `IS NOT NULL`, as already shown.

    If the constant is a range boundary, the comparison is folded to `=`. For example (using the same table as already defined):

    ```
    mysql> EXPLAIN SELECT * FROM t WHERE c >= 255;
    *************************** 1. row ***************************
               id: 1
      select_type: SIMPLE
            table: t
       partitions: NULL
             type: ALL
    possible_keys: NULL
              key: NULL
          key_len: NULL
              ref: NULL
             rows: 5
         filtered: 20.00
            Extra: Using where
    1 row in set, 1 warning (0.00 sec)

    mysql> SHOW WARNINGS;
    *************************** 1. row ***************************
      Level: Note
       Code: 1003
    Message: /* select#1 */ select `test`.`t`.`ti` AS `ti` from `test`.`t` where (`test`.`t`.`ti` = 255)
    1 row in set (0.00 sec)
    ```

  - **Floating- or fixed-point value.** If the constant is one of the decimal types (such as `DECIMAL`, `REAL`, `DOUBLE`, or `FLOAT`) and has a nonzero decimal portion, it cannot be equal; fold accordingly. For other comparisons, round up or down to an integer value according to the sign, then perform a range check and handle as already described for integer-integer comparisons.

    A `REAL` value that is too small to be represented as `DECIMAL` is rounded to .01 or -.01 depending on the sign, then handled as a `DECIMAL`.

  - **String types.** Try to interpret the string value as an integer type, then handle the comparison as between integer values. If this fails, attempt to handle the value as a `REAL`.
- **DECIMAL or REAL column.** Decimal types are compared with constants of the following types as described here:

- **Integer value.** Perform a range check against the column value's integer part. If no folding results, convert the constant to `DECIMAL` with the same number of decimal places as the column value, then check it as a `DECIMAL` (see next).
- **DECIMAL or REAL value.** Check for overflow (that is, whether the constant has more digits in its integer part than allowed for the column's decimal type). If so, fold.

  If the constant has more significant fractional digits than column's type, truncate the constant. If the comparison operator is `=` or `<>`, fold. If the operator is `>=` or `<=`, adjust the operator due to truncation. For example, if column's type is `DECIMAL(3,1)`, `SELECT * FROM t WHERE f >= 10.13` becomes `SELECT * FROM t WHERE f > 10.1`.

  If the constant has fewer decimal digits than the column's type, convert it to a constant with same number of digits. For underflow of a `REAL` value (that is, too few fractional digits to represent it), convert the constant to decimal 0.
- **String value.** If the value can be interpreted as an integer type, handle it as such. Otherwise, try to handle it as `REAL`.

- **FLOAT or DOUBLE column.** `FLOAT(m,n)` or `DOUBLE(m,n)` values compared with constants are handled as follows:

  If the value overflows the range of the column, fold.

  If the value has more than *n* decimals, truncate, compensating during folding. For `=` and `<>` comparisons, fold to `TRUE`, `FALSE`, or `IS [NOT] NULL` as described previously; for other operators, adjust the operator.

  If the value has more than `m` integer digits, fold.

Limitations. This optimization cannot be used in the following cases:

1. With comparisons using `BETWEEN` or `IN`.
2. With `BIT` columns or columns using date or time types.
3. During the preparation phase for a prepared statement, although it can be applied during the optimization phase when the prepared statement is actually executed. This due to the fact that, during statement preparation, the value of the constant is not yet known.

8.2.1.15 IS NULL Optimization

MySQL can perform the same optimization on *`col_name`* `IS NULL` that it can use for *`col_name`* = *`constant_value`*. For example, MySQL can use indexes and ranges to search for `NULL` with `IS NULL`.

Examples:

```
SELECT * FROM tbl_name WHERE key_col IS NULL;

SELECT * FROM tbl_name WHERE key_col <=> NULL;

SELECT * FROM tbl_name
  WHERE key_col=const1 OR key_col=const2 OR key_col IS NULL;
```

If a `WHERE` clause includes a *`col_name`* `IS NULL` condition for a column that is declared as `NOT NULL`, that expression is optimized away. This optimization does not occur in cases when the column might produce `NULL` anyway (for example, if it comes from a table on the right side of a `LEFT JOIN`).

MySQL can also optimize the combination *`col_name`* `= expr OR` *`col_name`* `IS NULL`, a form that is common in resolved subqueries. `EXPLAIN` shows `ref_or_null` when this optimization is used.

This optimization can handle one `IS NULL` for any key part.

Some examples of queries that are optimized, assuming that there is an index on columns `a` and `b` of table `t2`:

```
SELECT * FROM t1 WHERE t1.a=expr OR t1.a IS NULL;

SELECT * FROM t1, t2 WHERE t1.a=t2.a OR t2.a IS NULL;

SELECT * FROM t1, t2
  WHERE (t1.a=t2.a OR t2.a IS NULL) AND t2.b=t1.b;

SELECT * FROM t1, t2
  WHERE t1.a=t2.a AND (t2.b=t1.b OR t2.b IS NULL);

SELECT * FROM t1, t2
  WHERE (t1.a=t2.a AND t2.a IS NULL AND ...)
  OR (t1.a=t2.a AND t2.a IS NULL AND ...);
```

`ref_or_null` works by first doing a read on the reference key, and then a separate search for rows with a `NULL` key value.

The optimization can handle only one `IS NULL` level. In the following query, MySQL uses key lookups only on the expression `(t1.a=t2.a AND t2.a IS NULL)` and is not able to use the key part on `b`:

```
SELECT * FROM t1, t2
  WHERE (t1.a=t2.a AND t2.a IS NULL)
  OR (t1.b=t2.b AND t2.b IS NULL);
```

8.2.1.16 ORDER BY Optimization

This section describes when MySQL can use an index to satisfy an `ORDER BY` clause, the `filesort` operation used when an index cannot be used, and execution plan information available from the optimizer about `ORDER BY`.

An `ORDER BY` with and without `LIMIT` may return rows in different orders, as discussed in Section 8.2.1.19, “LIMIT Query Optimization”.

- Use of Indexes to Satisfy ORDER BY
- Use of filesort to Satisfy ORDER BY
- Influencing ORDER BY Optimization
- ORDER BY Execution Plan Information Available

Use of Indexes to Satisfy ORDER BY

In some cases, MySQL may use an index to satisfy an `ORDER BY` clause and avoid the extra sorting involved in performing a `filesort` operation.

The index may also be used even if the `ORDER BY` does not match the index exactly, as long as all unused portions of the index and all extra `ORDER BY` columns are constants in the `WHERE` clause. If the index does not contain all columns accessed by the query, the index is used only if index access is cheaper than other access methods.

Assuming that there is an index on `(key_part1, key_part2)`, the following queries may use the index to resolve the `ORDER BY` part. Whether the optimizer actually does so depends on whether reading the index is more efficient than a table scan if columns not in the index must also be read.

- In this query, the index on `(key_part1, key_part2)` enables the optimizer to avoid sorting:

```
SELECT * FROM t1
  ORDER BY key_part1, key_part2;
```

However, the query uses `SELECT *`, which may select more columns than *`key_part1`* and *`key_part2`*. In that case, scanning an entire index and looking up table rows to find columns not in the index may be more expensive than scanning the table and sorting the results. If so, the optimizer probably does not use the index. If `SELECT *` selects only the index columns, the index is used and sorting avoided.

If `t1` is an `InnoDB` table, the table primary key is implicitly part of the index, and the index can be used to resolve the `ORDER BY` for this query:

```
SELECT pk, key_part1, key_part2 FROM t1
  ORDER BY key_part1, key_part2;
```

- In this query, *`key_part1`* is constant, so all rows accessed through the index are in *`key_part2`* order, and an index on `(key_part1, key_part2)` avoids sorting if the `WHERE` clause is selective enough to make an index range scan cheaper than a table scan:

```
SELECT * FROM t1
  WHERE key_part1 = constant
  ORDER BY key_part2;
```

- In the next two queries, whether the index is used is similar to the same queries without `DESC` shown previously:

```
SELECT * FROM t1
  ORDER BY key_part1 DESC, key_part2 DESC;

SELECT * FROM t1
  WHERE key_part1 = constant
  ORDER BY key_part2 DESC;
```

- Two columns in an `ORDER BY` can sort in the same direction (both `ASC`, or both `DESC`) or in opposite directions (one `ASC`, one `DESC`). A condition for index use is that the index must have the same homogeneity, but need not have the same actual direction.

  If a query mixes `ASC` and `DESC`, the optimizer can use an index on the columns if the index also uses corresponding mixed ascending and descending columns:

```
SELECT * FROM t1
  ORDER BY key_part1 DESC, key_part2 ASC;
```

  The optimizer can use an index on (*`key_part1`*, *`key_part2`*) if *`key_part1`* is descending and *`key_part2`* is ascending. It can also use an index on those columns (with a backward scan) if *`key_part1`* is ascending and *`key_part2`* is descending. See Section 8.3.13, “Descending Indexes”.

- In the next two queries, *`key_part1`* is compared to a constant. The index is used if the `WHERE` clause is selective enough to make an index range scan cheaper than a table scan:

```
SELECT * FROM t1
  WHERE key_part1 > constant
  ORDER BY key_part1 ASC;

SELECT * FROM t1
  WHERE key_part1 < constant
  ORDER BY key_part1 DESC;
```

- In the next query, the `ORDER BY` does not name *`key_part1`*, but all rows selected have a constant *`key_part1`* value, so the index can still be used:

```
SELECT * FROM t1
  WHERE key_part1 = constant1 AND key_part2 > constant2
  ORDER BY key_part2;
```

In some cases, MySQL *cannot* use indexes to resolve the `ORDER BY`, although it may still use indexes to find the rows that match the `WHERE` clause. Examples:

- The query uses `ORDER BY` on different indexes:

```
SELECT * FROM t1 ORDER BY key1, key2;
```

- The query uses `ORDER BY` on nonconsecutive parts of an index:

```
SELECT * FROM t1 WHERE key2=constant ORDER BY key1_part1, key1_part3;
```

- The index used to fetch the rows differs from the one used in the `ORDER BY`:

```
SELECT * FROM t1 WHERE key2=constant ORDER BY key1;
```

- The query uses `ORDER BY` with an expression that includes terms other than the index column name:

```
SELECT * FROM t1 ORDER BY ABS(key);
SELECT * FROM t1 ORDER BY -key;
```

- The query joins many tables, and the columns in the `ORDER BY` are not all from the first nonconstant table that is used to retrieve rows. (This is the first table in the `EXPLAIN` output that does not have a `const` join type.)
- The query has different `ORDER BY` and `GROUP BY` expressions.
- There is an index on only a prefix of a column named in the `ORDER BY` clause. In this case, the index cannot be used to fully resolve the sort order. For example, if only the first 10 bytes of a `CHAR(20)` column are indexed, the index cannot distinguish values past the 10th byte and a `filesort` is needed.
- The index does not store rows in order. For example, this is true for a `HASH` index in a `MEMORY` table.

Availability of an index for sorting may be affected by the use of column aliases. Suppose that the column `t1.a` is indexed. In this statement, the name of the column in the select list is `a`. It refers to `t1.a`, as does the reference to `a` in the `ORDER BY`, so the index on `t1.a` can be used:

```
SELECT a FROM t1 ORDER BY a;
```

In this statement, the name of the column in the select list is also `a`, but it is the alias name. It refers to `ABS(a)`, as does the reference to `a` in the `ORDER BY`, so the index on `t1.a` cannot be used:

```
SELECT ABS(a) AS a FROM t1 ORDER BY a;
```

In the following statement, the `ORDER BY` refers to a name that is not the name of a column in the select list. But there is a column in `t1` named `a`, so the `ORDER BY` refers to `t1.a` and the index on `t1.a` can be used. (The resulting sort order may be completely different from the order for `ABS(a)`, of course.)

```
SELECT ABS(a) AS b FROM t1 ORDER BY a;
```

Previously (MySQL 5.7 and lower), `GROUP BY` sorted implicitly under certain conditions. In MySQL 8.0, that no longer occurs, so specifying `ORDER BY NULL` at the end to suppress implicit sorting (as was done previously) is no longer necessary. However, query results may differ from previous MySQL versions. To produce a given sort order, provide an `ORDER BY` clause.

Use of filesort to Satisfy ORDER BY

If an index cannot be used to satisfy an `ORDER BY` clause, MySQL performs a `filesort` operation that reads table rows and sorts them. A `filesort` constitutes an extra sorting phase in query execution.

To obtain memory for `filesort` operations, as of MySQL 8.0.12, the optimizer allocates memory buffers incrementally as needed, up to the size indicated by the `sort_buffer_size` system variable, rather than allocating a fixed amount of `sort_buffer_size` bytes up front, as was done prior to

MySQL 8.0.12. This enables users to set `sort_buffer_size` to larger values to speed up larger sorts, without concern for excessive memory use for small sorts. (This benefit may not occur for multiple concurrent sorts on Windows, which has a weak multithreaded `malloc`.)

A `filesort` operation uses temporary disk files as necessary if the result set is too large to fit in memory. Some types of queries are particularly suited to completely in-memory `filesort` operations. For example, the optimizer can use `filesort` to efficiently handle in memory, without temporary files, the `ORDER BY` operation for queries (and subqueries) of the following form:

```
SELECT ... FROM single_table ... ORDER BY non_index_column [DESC] LIMIT [M,]N;
```

Such queries are common in web applications that display only a few rows from a larger result set. Examples:

```
SELECT col1, ... FROM t1 ... ORDER BY name LIMIT 10;
SELECT col1, ... FROM t1 ... ORDER BY RAND() LIMIT 15;
```

Influencing ORDER BY Optimization

For slow `ORDER BY` queries for which `filesort` is not used, try lowering the `max_length_for_sort_data` system variable to a value that is appropriate to trigger a `filesort`. (A symptom of setting the value of this variable too high is a combination of high disk activity and low CPU activity.) This technique applies only before MySQL 8.0.20. As of 8.0.20, `max_length_for_sort_data` is deprecated due to optimizer changes that make it obsolete and of no effect.

To increase `ORDER BY` speed, check whether you can get MySQL to use indexes rather than an extra sorting phase. If this is not possible, try the following strategies:

- Increase the `sort_buffer_size` variable value. Ideally, the value should be large enough for the entire result set to fit in the sort buffer (to avoid writes to disk and merge passes).

  Take into account that the size of column values stored in the sort buffer is affected by the `max_sort_length` system variable value. For example, if tuples store values of long string columns and you increase the value of `max_sort_length`, the size of sort buffer tuples increases as well and may require you to increase `sort_buffer_size`.

  To monitor the number of merge passes (to merge temporary files), check the `Sort_merge_passes` status variable.

- Increase the `read_rnd_buffer_size` variable value so that more rows are read at a time.

- Change the `tmpdir` system variable to point to a dedicated file system with large amounts of free space. The variable value can list several paths that are used in round-robin fashion; you can use this feature to spread the load across several directories. Separate the paths by colon characters (`:`) on Unix and semicolon characters (`;`) on Windows. The paths should name directories in file systems located on different *physical* disks, not different partitions on the same disk.

ORDER BY Execution Plan Information Available

With `EXPLAIN` (see Section 8.8.1, “Optimizing Queries with EXPLAIN”), you can check whether MySQL can use indexes to resolve an `ORDER BY` clause:

- If the `Extra` column of `EXPLAIN` output does not contain `Using filesort`, the index is used and a `filesort` is not performed.

- If the `Extra` column of `EXPLAIN` output contains `Using filesort`, the index is not used and a `filesort` is performed.

In addition, if a `filesort` is performed, optimizer trace output includes a `filesort_summary` block. For example:

```
"filesort_summary": {
  "rows": 100,
  "examined_rows": 100,
  "number_of_tmp_files": 0,
  "peak_memory_used": 25192,
  "sort_mode": "<sort_key, packed_additional_fields>"
}
```

`peak_memory_used` indicates the maximum memory used at any one time during the sort. This is a value up to but not necessarily as large as the value of the `sort_buffer_size` system variable. Prior to MySQL 8.0.12, the output shows `sort_buffer_size` instead, indicating the value of `sort_buffer_size`. (Prior to MySQL 8.0.12, the optimizer always allocates `sort_buffer_size` bytes for the sort buffer. As of 8.0.12, the optimizer allocates sort-buffer memory incrementally, beginning with a small amount and adding more as necessary, up to `sort_buffer_size` bytes.)

The `sort_mode` value provides information about the contents of tuples in the sort buffer:

- `<sort_key, rowid>`: This indicates that sort buffer tuples are pairs that contain the sort key value and row ID of the original table row. Tuples are sorted by sort key value and the row ID is used to read the row from the table.
- `<sort_key, additional_fields>`: This indicates that sort buffer tuples contain the sort key value and columns referenced by the query. Tuples are sorted by sort key value and column values are read directly from the tuple.
- `<sort_key, packed_additional_fields>`: Like the previous variant, but the additional columns are packed tightly together instead of using a fixed-length encoding.

`EXPLAIN` does not distinguish whether the optimizer does or does not perform a `filesort` in memory. Use of an in-memory `filesort` can be seen in optimizer trace output. Look for `filesort_priority_queue_optimization`. For information about the optimizer trace, see MySQL Internals: Tracing the Optimizer.

8.2.1.17 GROUP BY Optimization

The most general way to satisfy a `GROUP BY` clause is to scan the whole table and create a new temporary table where all rows from each group are consecutive, and then use this temporary table to discover groups and apply aggregate functions (if any). In some cases, MySQL is able to do much better than that and avoid creation of temporary tables by using index access.

The most important preconditions for using indexes for `GROUP BY` are that all `GROUP BY` columns reference attributes from the same index, and that the index stores its keys in order (as is true, for example, for a `BTREE` index, but not for a `HASH` index). Whether use of temporary tables can be replaced by index access also depends on which parts of an index are used in a query, the conditions specified for these parts, and the selected aggregate functions.

There are two ways to execute a `GROUP BY` query through index access, as detailed in the following sections. The first method applies the grouping operation together with all range predicates (if any). The second method first performs a range scan, and then groups the resulting tuples.

- Loose Index Scan
- Tight Index Scan

Loose Index Scan can also be used in the absence of `GROUP BY` under some conditions. See Skip Scan Range Access Method.

Loose Index Scan

The most efficient way to process `GROUP BY` is when an index is used to directly retrieve the grouping columns. With this access method, MySQL uses the property of some index types that the keys are

ordered (for example, BTREE). This property enables use of lookup groups in an index without having to consider all keys in the index that satisfy all WHERE conditions. This access method considers only a fraction of the keys in an index, so it is called a *Loose Index Scan*. When there is no WHERE clause, a Loose Index Scan reads as many keys as the number of groups, which may be a much smaller number than that of all keys. If the WHERE clause contains range predicates (see the discussion of the range join type in Section 8.8.1, “Optimizing Queries with EXPLAIN”), a Loose Index Scan looks up the first key of each group that satisfies the range conditions, and again reads the smallest possible number of keys. This is possible under the following conditions:

- The query is over a single table.
- The GROUP BY names only columns that form a leftmost prefix of the index and no other columns. (If, instead of GROUP BY, the query has a DISTINCT clause, all distinct attributes refer to columns that form a leftmost prefix of the index.) For example, if a table t1 has an index on (c1,c2,c3), Loose Index Scan is applicable if the query has GROUP BY c1, c2. It is not applicable if the query has GROUP BY c2, c3 (the columns are not a leftmost prefix) or GROUP BY c1, c2, c4 (c4 is not in the index).
- The only aggregate functions used in the select list (if any) are MIN() and MAX(), and all of them refer to the same column. The column must be in the index and must immediately follow the columns in the GROUP BY.
- Any other parts of the index than those from the GROUP BY referenced in the query must be constants (that is, they must be referenced in equalities with constants), except for the argument of MIN() or MAX() functions.
- For columns in the index, full column values must be indexed, not just a prefix. For example, with c1 VARCHAR(20), INDEX (c1(10)), the index uses only a prefix of c1 values and cannot be used for Loose Index Scan.

If Loose Index Scan is applicable to a query, the EXPLAIN output shows Using index for group-by in the Extra column.

Assume that there is an index idx(c1,c2,c3) on table t1(c1,c2,c3,c4). The Loose Index Scan access method can be used for the following queries:

```
SELECT c1, c2 FROM t1 GROUP BY c1, c2;
SELECT DISTINCT c1, c2 FROM t1;
SELECT c1, MIN(c2) FROM t1 GROUP BY c1;
SELECT c1, c2 FROM t1 WHERE c1 < const GROUP BY c1, c2;
SELECT MAX(c3), MIN(c3), c1, c2 FROM t1 WHERE c2 > const GROUP BY c1, c2;
SELECT c2 FROM t1 WHERE c1 < const GROUP BY c1, c2;
SELECT c1, c2 FROM t1 WHERE c3 = const GROUP BY c1, c2;
```

The following queries cannot be executed with this quick select method, for the reasons given:

- There are aggregate functions other than MIN() or MAX():

  ```
  SELECT c1, SUM(c2) FROM t1 GROUP BY c1;
  ```

- The columns in the GROUP BY clause do not form a leftmost prefix of the index:

  ```
  SELECT c1, c2 FROM t1 GROUP BY c2, c3;
  ```

- The query refers to a part of a key that comes after the GROUP BY part, and for which there is no equality with a constant:

  ```
  SELECT c1, c3 FROM t1 GROUP BY c1, c2;
  ```

  Were the query to include WHERE c3 = *const*, Loose Index Scan could be used.

The Loose Index Scan access method can be applied to other forms of aggregate function references in the select list, in addition to the MIN() and MAX() references already supported:

- `AVG(DISTINCT)`, `SUM(DISTINCT)`, and `COUNT(DISTINCT)` are supported. `AVG(DISTINCT)` and `SUM(DISTINCT)` take a single argument. `COUNT(DISTINCT)` can have more than one column argument.
- There must be no `GROUP BY` or `DISTINCT` clause in the query.
- The Loose Index Scan limitations described previously still apply.

Assume that there is an index `idx(c1,c2,c3)` on table `t1(c1,c2,c3,c4)`. The Loose Index Scan access method can be used for the following queries:

```
SELECT COUNT(DISTINCT c1), SUM(DISTINCT c1) FROM t1;

SELECT COUNT(DISTINCT c1, c2), COUNT(DISTINCT c2, c1) FROM t1;
```

Tight Index Scan

A Tight Index Scan may be either a full index scan or a range index scan, depending on the query conditions.

When the conditions for a Loose Index Scan are not met, it still may be possible to avoid creation of temporary tables for `GROUP BY` queries. If there are range conditions in the `WHERE` clause, this method reads only the keys that satisfy these conditions. Otherwise, it performs an index scan. Because this method reads all keys in each range defined by the `WHERE` clause, or scans the whole index if there are no range conditions, it is called a *Tight Index Scan*. With a Tight Index Scan, the grouping operation is performed only after all keys that satisfy the range conditions have been found.

For this method to work, it is sufficient that there be a constant equality condition for all columns in a query referring to parts of the key coming before or in between parts of the `GROUP BY` key. The constants from the equality conditions fill in any “gaps” in the search keys so that it is possible to form complete prefixes of the index. These index prefixes then can be used for index lookups. If the `GROUP BY` result requires sorting, and it is possible to form search keys that are prefixes of the index, MySQL also avoids extra sorting operations because searching with prefixes in an ordered index already retrieves all the keys in order.

Assume that there is an index `idx(c1,c2,c3)` on table `t1(c1,c2,c3,c4)`. The following queries do not work with the Loose Index Scan access method described previously, but still work with the Tight Index Scan access method.

- There is a gap in the `GROUP BY`, but it is covered by the condition `c2 = 'a'`:

  ```
  SELECT c1, c2, c3 FROM t1 WHERE c2 = 'a' GROUP BY c1, c3;
  ```

- The `GROUP BY` does not begin with the first part of the key, but there is a condition that provides a constant for that part:

  ```
  SELECT c1, c2, c3 FROM t1 WHERE c1 = 'a' GROUP BY c2, c3;
  ```

8.2.1.18 DISTINCT Optimization

`DISTINCT` combined with `ORDER BY` needs a temporary table in many cases.

Because `DISTINCT` may use `GROUP BY`, learn how MySQL works with columns in `ORDER BY` or `HAVING` clauses that are not part of the selected columns. See Section 12.20.3, “MySQL Handling of GROUP BY”.

In most cases, a `DISTINCT` clause can be considered as a special case of `GROUP BY`. For example, the following two queries are equivalent:

```
SELECT DISTINCT c1, c2, c3 FROM t1
WHERE c1 > const;

SELECT c1, c2, c3 FROM t1
WHERE c1 > const GROUP BY c1, c2, c3;
```

Due to this equivalence, the optimizations applicable to `GROUP BY` queries can be also applied to queries with a `DISTINCT` clause. Thus, for more details on the optimization possibilities for `DISTINCT` queries, see Section 8.2.1.17, “GROUP BY Optimization”.

When combining `LIMIT row_count` with `DISTINCT`, MySQL stops as soon as it finds *`row_count`* unique rows.

If you do not use columns from all tables named in a query, MySQL stops scanning any unused tables as soon as it finds the first match. In the following case, assuming that `t1` is used before `t2` (which you can check with `EXPLAIN`), MySQL stops reading from `t2` (for any particular row in `t1`) when it finds the first row in `t2`:

```
SELECT DISTINCT t1.a FROM t1, t2 where t1.a=t2.a;
```

8.2.1.19 LIMIT Query Optimization

If you need only a specified number of rows from a result set, use a `LIMIT` clause in the query, rather than fetching the whole result set and throwing away the extra data.

MySQL sometimes optimizes a query that has a `LIMIT row_count` clause and no `HAVING` clause:

- If you select only a few rows with `LIMIT`, MySQL uses indexes in some cases when normally it would prefer to do a full table scan.
- If you combine `LIMIT row_count` with `ORDER BY`, MySQL stops sorting as soon as it has found the first *`row_count`* rows of the sorted result, rather than sorting the entire result. If ordering is done by using an index, this is very fast. If a filesort must be done, all rows that match the query without the `LIMIT` clause are selected, and most or all of them are sorted, before the first *`row_count`* are found. After the initial rows have been found, MySQL does not sort any remainder of the result set.

  One manifestation of this behavior is that an `ORDER BY` query with and without `LIMIT` may return rows in different order, as described later in this section.
- If you combine `LIMIT row_count` with `DISTINCT`, MySQL stops as soon as it finds *`row_count`* unique rows.
- In some cases, a `GROUP BY` can be resolved by reading the index in order (or doing a sort on the index), then calculating summaries until the index value changes. In this case, `LIMIT row_count` does not calculate any unnecessary `GROUP BY` values.
- As soon as MySQL has sent the required number of rows to the client, it aborts the query unless you are using `SQL_CALC_FOUND_ROWS`. In that case, the number of rows can be retrieved with `SELECT FOUND_ROWS()`. See Section 12.16, “Information Functions”.
- `LIMIT 0` quickly returns an empty set. This can be useful for checking the validity of a query. It can also be employed to obtain the types of the result columns within applications that use a MySQL API that makes result set metadata available. With the `mysql` client program, you can use the `--column-type-info` option to display result column types.
- If the server uses temporary tables to resolve a query, it uses the `LIMIT row_count` clause to calculate how much space is required.
- If an index is not used for `ORDER BY` but a `LIMIT` clause is also present, the optimizer may be able to avoid using a merge file and sort the rows in memory using an in-memory `filesort` operation.

If multiple rows have identical values in the `ORDER BY` columns, the server is free to return those rows in any order, and may do so differently depending on the overall execution plan. In other words, the sort order of those rows is nondeterministic with respect to the nonordered columns.

One factor that affects the execution plan is `LIMIT`, so an `ORDER BY` query with and without `LIMIT` may return rows in different orders. Consider this query, which is sorted by the `category` column but nondeterministic with respect to the `id` and `rating` columns:

```
mysql> SELECT * FROM ratings ORDER BY category;
+----+----------+--------+
| id | category | rating |
+----+----------+--------+
|  1 |        1 |    4.5 |
|  5 |        1 |    3.2 |
|  3 |        2 |    3.7 |
|  4 |        2 |    3.5 |
|  6 |        2 |    3.5 |
|  2 |        3 |    5.0 |
|  7 |        3 |    2.7 |
+----+----------+--------+
```

Including `LIMIT` may affect order of rows within each `category` value. For example, this is a valid query result:

```
mysql> SELECT * FROM ratings ORDER BY category LIMIT 5;
+----+----------+--------+
| id | category | rating |
+----+----------+--------+
|  1 |        1 |    4.5 |
|  5 |        1 |    3.2 |
|  4 |        2 |    3.5 |
|  3 |        2 |    3.7 |
|  6 |        2 |    3.5 |
+----+----------+--------+
```

In each case, the rows are sorted by the `ORDER BY` column, which is all that is required by the SQL standard.

If it is important to ensure the same row order with and without `LIMIT`, include additional columns in the `ORDER BY` clause to make the order deterministic. For example, if `id` values are unique, you can make rows for a given `category` value appear in `id` order by sorting like this:

```
mysql> SELECT * FROM ratings ORDER BY category, id;
+----+----------+--------+
| id | category | rating |
+----+----------+--------+
|  1 |        1 |    4.5 |
|  5 |        1 |    3.2 |
|  3 |        2 |    3.7 |
|  4 |        2 |    3.5 |
|  6 |        2 |    3.5 |
|  2 |        3 |    5.0 |
|  7 |        3 |    2.7 |
+----+----------+--------+

mysql> SELECT * FROM ratings ORDER BY category, id LIMIT 5;
+----+----------+--------+
| id | category | rating |
+----+----------+--------+
|  1 |        1 |    4.5 |
|  5 |        1 |    3.2 |
|  3 |        2 |    3.7 |
|  4 |        2 |    3.5 |
|  6 |        2 |    3.5 |
+----+----------+--------+
```

For a query with an `ORDER BY` or `GROUP BY` and a `LIMIT` clause, the optimizer tries to choose an ordered index by default when it appears doing so would speed up query execution. Prior to MySQL 8.0.21, there was no way to override this behavior, even in cases where using some other optimization might be faster. Beginning with MySQL 8.0.21, it is possible to turn off this optimization by setting the `optimizer_switch` system variable's `prefer_ordering_index` flag to `off`.

Example: First we create and populate a table `t` as shown here:

```
# Create and populate a table t:
```

```
mysql> CREATE TABLE t (
    ->     id1 BIGINT NOT NULL,
    ->     id2 BIGINT NOT NULL,
    ->     c1 VARCHAR(50) NOT NULL,
    ->     c2 VARCHAR(50) NOT NULL,
    ->  PRIMARY KEY (id1),
    ->  INDEX i (id2, c1)
    -> );

# [Insert some rows into table t - not shown]
```

Verify that the `prefer_ordering_index` flag is enabled:

```
mysql> SELECT @@optimizer_switch LIKE '%prefer_ordering_index=on%';
+-----------------------------------------------------+
| @@optimizer_switch LIKE '%prefer_ordering_index=on%' |
+-----------------------------------------------------+
|                                                   1 |
+-----------------------------------------------------+
```

Since the following query has a `LIMIT` clause, we expect it to use an ordered index, if possible. In this case, as we can see from the `EXPLAIN` output, it uses the table's primary key.

```
mysql> EXPLAIN SELECT c2 FROM t
    ->     WHERE id2 > 3
    ->     ORDER BY id1 ASC LIMIT 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
   partitions: NULL
         type: index
possible_keys: i
          key: PRIMARY
      key_len: 8
          ref: NULL
         rows: 2
     filtered: 70.00
        Extra: Using where
```

Now we disable the `prefer_ordering_index` flag, and re-run the same query; this time it uses the index `i` (which includes the `id2` column used in the `WHERE` clause), and a filesort:

```
mysql> SET optimizer_switch = "prefer_ordering_index=off";

mysql> EXPLAIN SELECT c2 FROM t
    ->     WHERE id2 > 3
    ->     ORDER BY id1 ASC LIMIT 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
   partitions: NULL
         type: range
possible_keys: i
          key: i
      key_len: 8
          ref: NULL
         rows: 14
     filtered: 100.00
        Extra: Using index condition; Using filesort
```

See also Section 8.9.2, “Switchable Optimizations”.

8.2.1.20 Function Call Optimization

MySQL functions are tagged internally as deterministic or nondeterministic. A function is nondeterministic if, given fixed values for its arguments, it can return different results for different invocations. Examples of nondeterministic functions: `RAND()`, `UUID()`.

If a function is tagged nondeterministic, a reference to it in a `WHERE` clause is evaluated for every row (when selecting from one table) or combination of rows (when selecting from a multiple-table join).

MySQL also determines when to evaluate functions based on types of arguments, whether the arguments are table columns or constant values. A deterministic function that takes a table column as argument must be evaluated whenever that column changes value.

Nondeterministic functions may affect query performance. For example, some optimizations may not be available, or more locking might be required. The following discussion uses `RAND()` but applies to other nondeterministic functions as well.

Suppose that a table `t` has this definition:

```
CREATE TABLE t (id INT NOT NULL PRIMARY KEY, col_a VARCHAR(100));
```

Consider these two queries:

```
SELECT * FROM t WHERE id = POW(1,2);
SELECT * FROM t WHERE id = FLOOR(1 + RAND() * 49);
```

Both queries appear to use a primary key lookup because of the equality comparison against the primary key, but that is true only for the first of them:

- The first query always produces a maximum of one row because `POW()` with constant arguments is a constant value and is used for index lookup.
- The second query contains an expression that uses the nondeterministic function `RAND()`, which is not constant in the query but in fact has a new value for every row of table `t`. Consequently, the query reads every row of the table, evaluates the predicate for each row, and outputs all rows for which the primary key matches the random value. This might be zero, one, or multiple rows, depending on the `id` column values and the values in the `RAND()` sequence.

The effects of nondeterminism are not limited to `SELECT` statements. This `UPDATE` statement uses a nondeterministic function to select rows to be modified:

```
UPDATE t SET col_a = some_expr WHERE id = FLOOR(1 + RAND() * 49);
```

Presumably the intent is to update at most a single row for which the primary key matches the expression. However, it might update zero, one, or multiple rows, depending on the `id` column values and the values in the `RAND()` sequence.

The behavior just described has implications for performance and replication:

- Because a nondeterministic function does not produce a constant value, the optimizer cannot use strategies that might otherwise be applicable, such as index lookups. The result may be a table scan.
- `InnoDB` might escalate to a range-key lock rather than taking a single row lock for one matching row.
- Updates that do not execute deterministically are unsafe for replication.

The difficulties stem from the fact that the `RAND()` function is evaluated once for every row of the table. To avoid multiple function evaluations, use one of these techniques:

- Move the expression containing the nondeterministic function to a separate statement, saving the value in a variable. In the original statement, replace the expression with a reference to the variable, which the optimizer can treat as a constant value:

  ```
  SET @keyval = FLOOR(1 + RAND() * 49);
  UPDATE t SET col_a = some_expr WHERE id = @keyval;
  ```

- Assign the random value to a variable in a derived table. This technique causes the variable to be assigned a value, once, prior to its use in the comparison in the `WHERE` clause:

  ```
  UPDATE /*+ NO_MERGE(dt) */ t, (SELECT FLOOR(1 + RAND() * 49) AS r) AS dt
  ```

```
SET col_a = some_expr WHERE id = dt.r;
```

As mentioned previously, a nondeterministic expression in the `WHERE` clause might prevent optimizations and result in a table scan. However, it may be possible to partially optimize the `WHERE` clause if other expressions are deterministic. For example:

```
SELECT * FROM t WHERE partial_key=5 AND some_column=RAND();
```

If the optimizer can use `partial_key` to reduce the set of rows selected, `RAND()` is executed fewer times, which diminishes the effect of nondeterminism on optimization.

8.2.1.21 Window Function Optimization

Window functions affect the strategies the optimizer considers:

- Derived table merging for a subquery is disabled if the subquery has window functions. The subquery is always materialized.
- Semijoins are not applicable to window function optimization because semijoins apply to subqueries in `WHERE` and `JOIN ... ON`, which cannot contain window functions.
- The optimizer processes multiple windows that have the same ordering requirements in sequence, so sorting can be skipped for windows following the first one.
- The optimizer makes no attempt to merge windows that could be evaluated in a single step (for example, when multiple `OVER` clauses contain identical window definitions). The workaround is to define the window in a `WINDOW` clause and refer to the window name in the `OVER` clauses.

An aggregate function not used as a window function is aggregated in the outermost possible query. For example, in this query, MySQL sees that `COUNT(t1.b)` is something that cannot exist in the outer query because of its placement in the `WHERE` clause:

```
SELECT * FROM t1 WHERE t1.a = (SELECT COUNT(t1.b) FROM t2);
```

Consequently, MySQL aggregates inside the subquery, treating `t1.b` as a constant and returning the count of rows of `t2`.

Replacing `WHERE` with `HAVING` results in an error:

```
mysql> SELECT * FROM t1 HAVING t1.a = (SELECT COUNT(t1.b) FROM t2);
ERROR 1140 (42000): In aggregated query without GROUP BY, expression #1
of SELECT list contains nonaggregated column 'test.t1.a'; this is
incompatible with sql_mode=only_full_group_by
```

The error occurs because `COUNT(t1.b)` can exist in the `HAVING`, and so makes the outer query aggregated.

Window functions (including aggregate functions used as window functions) do not have the preceding complexity. They always aggregate in the subquery where they are written, never in the outer query.

Window function evaluation may be affected by the value of the `windowing_use_high_precision` system variable, which determines whether to compute window operations without loss of precision. By default, `windowing_use_high_precision` is enabled.

For some moving frame aggregates, the inverse aggregate function can be applied to remove values from the aggregate. This can improve performance but possibly with a loss of precision. For example, adding a very small floating-point value to a very large value causes the very small value to be “hidden” by the large value. When inverting the large value later, the effect of the small value is lost.

Loss of precision due to inverse aggregation is a factor only for operations on floating-point (approximate-value) data types. For other types, inverse aggregation is safe; this includes `DECIMAL`, which permits a fractional part but is an exact-value type.

For faster execution, MySQL always uses inverse aggregation when it is safe:

- For floating-point values, inverse aggregation is not always safe and might result in loss of precision. The default is to avoid inverse aggregation, which is slower but preserves precision. If it is permissible to sacrifice safety for speed, `windowing_use_high_precision` can be disabled to permit inverse aggregation.
- For nonfloating-point data types, inverse aggregation is always safe and is used regardless of the `windowing_use_high_precision` value.
- `windowing_use_high_precision` has no effect on `MIN()` and `MAX()`, which do not use inverse aggregation in any case.

For evaluation of the variance functions `STDDEV_POP()`, `STDDEV_SAMP()`, `VAR_POP()`, `VAR_SAMP()`, and their synonyms, evaluation can occur in optimized mode or default mode. Optimized mode may produce slightly different results in the last significant digits. If such differences are permissible, `windowing_use_high_precision` can be disabled to permit optimized mode.

For `EXPLAIN`, windowing execution plan information is too extensive to display in traditional output format. To see windowing information, use `EXPLAIN FORMAT=JSON` and look for the `windowing` element.

8.2.1.22 Row Constructor Expression Optimization

Row constructors permit simultaneous comparisons of multiple values. For example, these two statements are semantically equivalent:

```
SELECT * FROM t1 WHERE (column1,column2) = (1,1);
SELECT * FROM t1 WHERE column1 = 1 AND column2 = 1;
```

In addition, the optimizer handles both expressions the same way.

The optimizer is less likely to use available indexes if the row constructor columns do not cover the prefix of an index. Consider the following table, which has a primary key on `(c1, c2, c3)`:

```
CREATE TABLE t1 (
  c1 INT, c2 INT, c3 INT, c4 CHAR(100),
  PRIMARY KEY(c1,c2,c3)
);
```

In this query, the `WHERE` clause uses all columns in the index. However, the row constructor itself does not cover an index prefix, with the result that the optimizer uses only `c1` (`key_len=4`, the size of `c1`):

```
mysql> EXPLAIN SELECT * FROM t1
       WHERE c1=1 AND (c2,c3) > (1,1)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 3
     filtered: 100.00
        Extra: Using where
```

In such cases, rewriting the row constructor expression using an equivalent nonconstructor expression may result in more complete index use. For the given query, the row constructor and equivalent nonconstructor expressions are:

```
(c2,c3) > (1,1)
```

```
c2 > 1 OR ((c2 = 1) AND (c3 > 1))
```

Rewriting the query to use the nonconstructor expression results in the optimizer using all three columns in the index (`key_len=12`):

```
mysql> EXPLAIN SELECT * FROM t1
       WHERE c1 = 1 AND (c2 > 1 OR ((c2 = 1) AND (c3 > 1)))\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 12
          ref: NULL
         rows: 3
     filtered: 100.00
        Extra: Using where
```

Thus, for better results, avoid mixing row constructors with `AND`/`OR` expressions. Use one or the other.

Under certain conditions, the optimizer can apply the range access method to `IN()` expressions that have row constructor arguments. See Range Optimization of Row Constructor Expressions.

8.2.1.23 Avoiding Full Table Scans

The output from `EXPLAIN` shows `ALL` in the `type` column when MySQL uses a full table scan to resolve a query. This usually happens under the following conditions:

- The table is so small that it is faster to perform a table scan than to bother with a key lookup. This is common for tables with fewer than 10 rows and a short row length.
- There are no usable restrictions in the `ON` or `WHERE` clause for indexed columns.
- You are comparing indexed columns with constant values and MySQL has calculated (based on the index tree) that the constants cover too large a part of the table and that a table scan would be faster. See Section 8.2.1.1, “WHERE Clause Optimization”.
- You are using a key with low cardinality (many rows match the key value) through another column. In this case, MySQL assumes that by using the key probably requires many key lookups and that a table scan would be faster.

For small tables, a table scan often is appropriate and the performance impact is negligible. For large tables, try the following techniques to avoid having the optimizer incorrectly choose a table scan:

- Use `ANALYZE TABLE` *`tbl_name`* to update the key distributions for the scanned table. See Section 13.7.3.1, “ANALYZE TABLE Statement”.
- Use `FORCE INDEX` for the scanned table to tell MySQL that table scans are very expensive compared to using the given index:

  ```
  SELECT * FROM t1, t2 FORCE INDEX (index_for_column)
    WHERE t1.col_name=t2.col_name;
  ```

  See Section 8.9.4, “Index Hints”.
- Start `mysqld` with the `--max-seeks-for-key=1000` option or use `SET max_seeks_for_key=1000` to tell the optimizer to assume that no key scan causes more than 1,000 key seeks. See Section 5.1.8, “Server System Variables”.

8.2.2 Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions

The MySQL query optimizer has different strategies available to evaluate subqueries:

- For a subquery used with an `IN`, `= ANY`, or `EXISTS` predicate, the optimizer has these choices:
  - Semijoin
  - Materialization
  - `EXISTS` strategy
- For a subquery used with a `NOT IN`, `<> ALL` or `NOT EXISTS` predicate, the optimizer has these choices:
  - Materialization
  - `EXISTS` strategy

For a derived table, the optimizer has these choices (which also apply to view references and common table expressions):

- Merge the derived table into the outer query block
- Materialize the derived table to an internal temporary table

The following discussion provides more information about the preceding optimization strategies.

Note

A limitation on `UPDATE` and `DELETE` statements that use a subquery to modify a single table is that the optimizer does not use semijoin or materialization subquery optimizations. As a workaround, try rewriting them as multiple-table `UPDATE` and `DELETE` statements that use a join rather than a subquery.

8.2.2.1 Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations

A semijoin is a preparation-time transformation that enables multiple execution strategies such as table pullout, duplicate weedout, first match, loose scan, and materialization. The optimizer uses semijoin strategies to improve subquery execution, as described in this section.

For an inner join between two tables, the join returns a row from one table as many times as there are matches in the other table. But for some questions, the only information that matters is whether there is a match, not the number of matches. Suppose that there are tables named `class` and `roster` that list classes in a course curriculum and class rosters (students enrolled in each class), respectively. To list the classes that actually have students enrolled, you could use this join:

```
SELECT class.class_num, class.class_name
    FROM class
    INNER JOIN roster
    WHERE class.class_num = roster.class_num;
```

However, the result lists each class once for each enrolled student. For the question being asked, this is unnecessary duplication of information.

Assuming that `class_num` is a primary key in the `class` table, duplicate suppression is possible by using `SELECT DISTINCT`, but it is inefficient to generate all matching rows first only to eliminate duplicates later.

The same duplicate-free result can be obtained by using a subquery:

```
SELECT class_num, class_name
    FROM class
    WHERE class_num IN
```

```
        (SELECT class_num FROM roster);
```

Here, the optimizer can recognize that the `IN` clause requires the subquery to return only one instance of each class number from the `roster` table. In this case, the query can use a *semijoin*; that is, an operation that returns only one instance of each row in `class` that is matched by rows in `roster`.

The following statement, which contains an `EXISTS` subquery predicate, is equivalent to the previous statement containing an `IN` subquery predicate:

```
SELECT class_num, class_name
    FROM class
    WHERE EXISTS
        (SELECT * FROM roster WHERE class.class_num = roster.class_num);
```

In MySQL 8.0.16 and later, any statement with an `EXISTS` subquery predicate is subject to the same semijoin transforms as a statement with an equivalent `IN` subquery predicate.

Beginning with MySQL 8.0.17, the following subqueries are transformed into antijoins:

- `NOT IN (SELECT ... FROM ...)`
- `NOT EXISTS (SELECT ... FROM ...)`.
- `IN (SELECT ... FROM ...) IS NOT TRUE`
- `EXISTS (SELECT ... FROM ...) IS NOT TRUE`.
- `IN (SELECT ... FROM ...) IS FALSE`
- `EXISTS (SELECT ... FROM ...) IS FALSE`.

In short, any negation of a subquery of the form `IN (SELECT ... FROM ...)` or `EXISTS (SELECT ... FROM ...)` is transformed into an antijoin.

An antijoin is an operation that returns only rows for which there is no match. Consider the query shown here:

```
SELECT class_num, class_name
    FROM class
    WHERE class_num NOT IN
        (SELECT class_num FROM roster);
```

This query is rewritten internally as the antijoin `SELECT class_num, class_name FROM class ANTIJOIN roster ON class_num`, which returns one instance of each row in `class` that is *not* matched by any rows in `roster`. This means that, for each row in `class`, as soon as a match is found in `roster`, the row in `class` can be discarded.

Antijoin transformations cannot in most cases be applied if the expressions being compared are nullable. An exception to this rule is that `(... NOT IN (SELECT ...)) IS NOT FALSE` and its equivalent `(... IN (SELECT ...)) IS NOT TRUE` can be transformed into antijoins.

Outer join and inner join syntax is permitted in the outer query specification, and table references may be base tables, derived tables, view references, or common table expressions.

In MySQL, a subquery must satisfy these criteria to be handled as a semijoin (or, in MySQL 8.0.17 and later, an antijoin if `NOT` modifies the subquery):

- It must be part of an `IN`, `= ANY`, or `EXISTS` predicate that appears at the top level of the `WHERE` or `ON` clause, possibly as a term in an `AND` expression. For example:

  ```
  SELECT ...
      FROM ot1, ...
      WHERE (oe1, ...) IN
          (SELECT ie1, ... FROM it1, ... WHERE ...);
  ```

Here, `ot_i` and `it_i` represent tables in the outer and inner parts of the query, and `oe_i` and `ie_i` represent expressions that refer to columns in the outer and inner tables.

In MySQL 8.0.17 and later, the subquery can also be the argument to an expression modified by `NOT`, `IS [NOT] TRUE`, or `IS [NOT] FALSE`.

- It must be a single `SELECT` without `UNION` constructs.
- It must not contain a `HAVING` clause.
- It must not contain any aggregate functions (whether it is explicitly or implicitly grouped).
- It must not have a `LIMIT` clause.
- The statement must not use the `STRAIGHT_JOIN` join type in the outer query.
- The `STRAIGHT_JOIN` modifier must not be present.
- The number of outer and inner tables together must be less than the maximum number of tables permitted in a join.
- The subquery may be correlated or uncorrelated. In MySQL 8.0.16 and later, decorrelation looks at trivially correlated predicates in the `WHERE` clause of a subquery used as the argument to `EXISTS`, and makes it possible to optimize it as if it was used within `IN (SELECT b FROM ...)`. The term *trivially correlated* means that the predicate is an equality predicate, that it is the sole predicate in the `WHERE` clause (or is combined with `AND`), and that one operand is from a table referenced in the subquery and the other operand is from the outer query block.
- The `DISTINCT` keyword is permitted but ignored. Semijoin strategies automatically handle duplicate removal.
- A `GROUP BY` clause is permitted but ignored, unless the subquery also contains one or more aggregate functions.
- An `ORDER BY` clause is permitted but ignored, since ordering is irrelevant to the evaluation of semijoin strategies.

If a subquery meets the preceding criteria, MySQL converts it to a semijoin (or, in MySQL 8.0.17 or later, an antijoin if applicable) and makes a cost-based choice from these strategies:

- Convert the subquery to a join, or use table pullout and run the query as an inner join between subquery tables and outer tables. Table pullout pulls a table out from the subquery to the outer query.
- *Duplicate Weedout*: Run the semijoin as if it was a join and remove duplicate records using a temporary table.
- *FirstMatch*: When scanning the inner tables for row combinations and there are multiple instances of a given value group, choose one rather than returning them all. This "shortcuts" scanning and eliminates production of unnecessary rows.
- *LooseScan*: Scan a subquery table using an index that enables a single value to be chosen from each subquery's value group.
- Materialize the subquery into an indexed temporary table that is used to perform a join, where the index is used to remove duplicates. The index might also be used later for lookups when joining the temporary table with the outer tables; if not, the table is scanned. For more information about materialization, see Section 8.2.2.2, “Optimizing Subqueries with Materialization”.

Each of these strategies can be enabled or disabled using the following `optimizer_switch` system variable flags:

- The `semijoin` flag controls whether semijoins are used. Starting with MySQL 8.0.17, this also applies to antijoins.
- If `semijoin` is enabled, the `firstmatch`, `loosescan`, `duplicateweedout`, and `materialization` flags enable finer control over the permitted semijoin strategies.
- If the `duplicateweedout` semijoin strategy is disabled, it is not used unless all other applicable strategies are also disabled.
- If `duplicateweedout` is disabled, on occasion the optimizer may generate a query plan that is far from optimal. This occurs due to heuristic pruning during greedy search, which can be avoided by setting `optimizer_prune_level=0`.

These flags are enabled by default. See Section 8.9.2, “Switchable Optimizations”.

The optimizer minimizes differences in handling of views and derived tables. This affects queries that use the `STRAIGHT_JOIN` modifier and a view with an `IN` subquery that can be converted to a semijoin. The following query illustrates this because the change in processing causes a change in transformation, and thus a different execution strategy:

```
CREATE VIEW v AS
SELECT *
FROM t1
WHERE a IN (SELECT b
            FROM t2);

SELECT STRAIGHT_JOIN *
FROM t3 JOIN v ON t3.x = v.a;
```

The optimizer first looks at the view and converts the `IN` subquery to a semijoin, then checks whether it is possible to merge the view into the outer query. Because the `STRAIGHT_JOIN` modifier in the outer query prevents semijoin, the optimizer refuses the merge, causing derived table evaluation using a materialized table.

`EXPLAIN` output indicates the use of semijoin strategies as follows:

- For extended `EXPLAIN` output, the text displayed by a following `SHOW WARNINGS` shows the rewritten query, which displays the semijoin structure. (See Section 8.8.3, “Extended EXPLAIN Output Format”.) From this you can get an idea about which tables were pulled out of the semijoin. If a subquery was converted to a semijoin, you should see that the subquery predicate is gone and its tables and `WHERE` clause were merged into the outer query join list and `WHERE` clause.
- Temporary table use for Duplicate Weedout is indicated by `Start temporary` and `End temporary` in the `Extra` column. Tables that were not pulled out and are in the range of `EXPLAIN` output rows covered by `Start temporary` and `End temporary` have their `rowid` in the temporary table.
- `FirstMatch(`*`tbl_name`*`)` in the `Extra` column indicates join shortcutting.
- `LooseScan(`*`m`*`..`*`n`*`)` in the `Extra` column indicates use of the LooseScan strategy. *`m`* and *`n`* are key part numbers.
- Temporary table use for materialization is indicated by rows with a `select_type` value of `MATERIALIZED` and rows with a `table` value of `<subquery`*`N`*`>`.

In MySQL 8.0.21 and later, a semijoin transformation can also be applied to a single-table `UPDATE` or `DELETE` statement that uses a `[NOT] IN` or `[NOT] EXISTS` subquery predicate, provided that the statement does not use `ORDER BY` or `LIMIT`, and that semijoin transformations are allowed by an optimizer hint or by the `optimizer_switch` setting.

8.2.2.2 Optimizing Subqueries with Materialization

The optimizer uses materialization to enable more efficient subquery processing. Materialization speeds up query execution by generating a subquery result as a temporary table, normally in memory.

The first time MySQL needs the subquery result, it materializes that result into a temporary table. Any subsequent time the result is needed, MySQL refers again to the temporary table. The optimizer may index the table with a hash index to make lookups fast and inexpensive. The index contains unique values to eliminate duplicates and make the table smaller.

Subquery materialization uses an in-memory temporary table when possible, falling back to on-disk storage if the table becomes too large. See Section 8.4.4, “Internal Temporary Table Use in MySQL”.

If materialization is not used, the optimizer sometimes rewrites a noncorrelated subquery as a correlated subquery. For example, the following `IN` subquery is noncorrelated (*`where_condition`* involves only columns from `t2` and not `t1`):

```
SELECT * FROM t1
WHERE t1.a IN (SELECT t2.b FROM t2 WHERE where_condition);
```

The optimizer might rewrite this as an `EXISTS` correlated subquery:

```
SELECT * FROM t1
WHERE EXISTS (SELECT t2.b FROM t2 WHERE where_condition AND t1.a=t2.b);
```

Subquery materialization using a temporary table avoids such rewrites and makes it possible to execute the subquery only once rather than once per row of the outer query.

For subquery materialization to be used in MySQL, the `optimizer_switch` system variable `materialization` flag must be enabled. (See Section 8.9.2, “Switchable Optimizations”.) With the `materialization` flag enabled, materialization applies to subquery predicates that appear anywhere (in the select list, `WHERE`, `ON`, `GROUP BY`, `HAVING`, or `ORDER BY`), for predicates that fall into any of these use cases:

- The predicate has this form, when no outer expression *`oe_i`* or inner expression *`ie_i`* is nullable. *`N`* is 1 or larger.

  ```
  (oe_1, oe_2, ..., oe_N) [NOT] IN (SELECT ie_1, i_2, ..., ie_N ...)
  ```

- The predicate has this form, when there is a single outer expression *`oe`* and inner expression *`ie`*. The expressions can be nullable.

  ```
  oe [NOT] IN (SELECT ie ...)
  ```

- The predicate is `IN` or `NOT IN` and a result of `UNKNOWN` (`NULL`) has the same meaning as a result of `FALSE`.

The following examples illustrate how the requirement for equivalence of `UNKNOWN` and `FALSE` predicate evaluation affects whether subquery materialization can be used. Assume that *`where_condition`* involves columns only from `t2` and not `t1` so that the subquery is noncorrelated.

This query is subject to materialization:

```
SELECT * FROM t1
WHERE t1.a IN (SELECT t2.b FROM t2 WHERE where_condition);
```

Here, it does not matter whether the `IN` predicate returns `UNKNOWN` or `FALSE`. Either way, the row from `t1` is not included in the query result.

An example where subquery materialization is not used is the following query, where `t2.b` is a nullable column:

```
SELECT * FROM t1
WHERE (t1.a,t1.b) NOT IN (SELECT t2.a,t2.b FROM t2
                          WHERE where_condition);
```

The following restrictions apply to the use of subquery materialization:

- The types of the inner and outer expressions must match. For example, the optimizer might be able to use materialization if both expressions are integer or both are decimal, but cannot if one expression is integer and the other is decimal.
- The inner expression cannot be a `BLOB`.

Use of `EXPLAIN` with a query provides some indication of whether the optimizer uses subquery materialization:

- Compared to query execution that does not use materialization, `select_type` may change from `DEPENDENT SUBQUERY` to `SUBQUERY`. This indicates that, for a subquery that would be executed once per outer row, materialization enables the subquery to be executed just once.
- For extended `EXPLAIN` output, the text displayed by a following `SHOW WARNINGS` includes `materialize` and `materialized-subquery`.

In MySQL 8.0.21 and later, MySQL can also apply subquery materialization to a single-table `UPDATE` or `DELETE` statement that uses a `[NOT] IN` or `[NOT] EXISTS` subquery predicate, provided that the statement does not use `ORDER BY` or `LIMIT`, and that subquery materialization is allowed by an optimizer hint or by the `optimizer_switch` setting.

8.2.2.3 Optimizing Subqueries with the EXISTS Strategy

Certain optimizations are applicable to comparisons that use the `IN` (or `=ANY`) operator to test subquery results. This section discusses these optimizations, particularly with regard to the challenges that `NULL` values present. The last part of the discussion suggests how you can help the optimizer.

Consider the following subquery comparison:

```
outer_expr IN (SELECT inner_expr FROM ... WHERE subquery_where)
```

MySQL evaluates queries “from outside to inside.” That is, it first obtains the value of the outer expression *`outer_expr`*, and then runs the subquery and captures the rows that it produces.

A very useful optimization is to “inform” the subquery that the only rows of interest are those where the inner expression *`inner_expr`* is equal to *`outer_expr`*. This is done by pushing down an appropriate equality into the subquery's `WHERE` clause to make it more restrictive. The converted comparison looks like this:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where AND outer_expr=inner_expr)
```

After the conversion, MySQL can use the pushed-down equality to limit the number of rows it must examine to evaluate the subquery.

More generally, a comparison of *`N`* values to a subquery that returns *`N`*-value rows is subject to the same conversion. If *`oe_i`* and *`ie_i`* represent corresponding outer and inner expression values, this subquery comparison:

```
(oe_1, ..., oe_N) IN
  (SELECT ie_1, ..., ie_N FROM ... WHERE subquery_where)
```

Becomes:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where
                          AND oe_1 = ie_1
                          AND ...
                          AND oe_N = ie_N)
```

For simplicity, the following discussion assumes a single pair of outer and inner expression values.

The “pushdown” strategy just described works if either of these conditions is true:

- *outer_expr* and *inner_expr* cannot be NULL.
- You need not distinguish NULL from FALSE subquery results. If the subquery is a part of an OR or AND expression in the WHERE clause, MySQL assumes that you do not care. Another instance where the optimizer notices that NULL and FALSE subquery results need not be distinguished is this construct:

  ```
  ... WHERE outer_expr IN (subquery)
  ```

  In this case, the WHERE clause rejects the row whether IN (*subquery*) returns NULL or FALSE.

Suppose that *outer_expr* is known to be a non-NULL value but the subquery does not produce a row such that *outer_expr* = *inner_expr*. Then *outer_expr* IN (SELECT ...) evaluates as follows:

- NULL, if the SELECT produces any row where *inner_expr* is NULL
- FALSE, if the SELECT produces only non-NULL values or produces nothing

In this situation, the approach of looking for rows with *outer_expr* = *inner_expr* is no longer valid. It is necessary to look for such rows, but if none are found, also look for rows where *inner_expr* is NULL. Roughly speaking, the subquery can be converted to something like this:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where AND
        (outer_expr=inner_expr OR inner_expr IS NULL))
```

The need to evaluate the extra IS NULL condition is why MySQL has the ref_or_null access method:

```
mysql> EXPLAIN
       SELECT outer_expr IN (SELECT t2.maybe_null_key
                             FROM t2, t3 WHERE ...)
       FROM t1;
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: t1
...
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: t2
         type: ref_or_null
possible_keys: maybe_null_key
          key: maybe_null_key
      key_len: 5
          ref: func
         rows: 2
        Extra: Using where; Using index
...
```

The unique_subquery and index_subquery subquery-specific access methods also have “or NULL” variants.

The additional OR ... IS NULL condition makes query execution slightly more complicated (and some optimizations within the subquery become inapplicable), but generally this is tolerable.

The situation is much worse when *outer_expr* can be NULL. According to the SQL interpretation of NULL as “unknown value,” NULL IN (SELECT *inner_expr* ...) should evaluate to:

- NULL, if the SELECT produces any rows
- FALSE, if the SELECT produces no rows

For proper evaluation, it is necessary to be able to check whether the SELECT has produced any rows at all, so *outer_expr* = *inner_expr* cannot be pushed down into the subquery. This is a problem because many real world subqueries become very slow unless the equality can be pushed down.

Essentially, there must be different ways to execute the subquery depending on the value of *`outer_expr`*.

The optimizer chooses SQL compliance over speed, so it accounts for the possibility that *`outer_expr`* might be `NULL`:

- If *`outer_expr`* is `NULL`, to evaluate the following expression, it is necessary to execute the `SELECT` to determine whether it produces any rows:

  ```
  NULL IN (SELECT inner_expr FROM ... WHERE subquery_where)
  ```

  It is necessary to execute the original `SELECT` here, without any pushed-down equalities of the kind mentioned previously.

- On the other hand, when *`outer_expr`* is not `NULL`, it is absolutely essential that this comparison:

  ```
  outer_expr IN (SELECT inner_expr FROM ... WHERE subquery_where)
  ```

  Be converted to this expression that uses a pushed-down condition:

  ```
  EXISTS (SELECT 1 FROM ... WHERE subquery_where AND outer_expr=inner_expr)
  ```

  Without this conversion, subqueries are slow.

To solve the dilemma of whether or not to push down conditions into the subquery, the conditions are wrapped within “trigger” functions. Thus, an expression of the following form:

```
outer_expr IN (SELECT inner_expr FROM ... WHERE subquery_where)
```

Is converted into:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where
                          AND trigcond(outer_expr=inner_expr))
```

More generally, if the subquery comparison is based on several pairs of outer and inner expressions, the conversion takes this comparison:

```
(oe_1, ..., oe_N) IN (SELECT ie_1, ..., ie_N FROM ... WHERE subquery_where)
```

And converts it to this expression:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where
                          AND trigcond(oe_1=ie_1)
                          AND ...
                          AND trigcond(oe_N=ie_N)
       )
```

Each `trigcond(`*`X`*`)` is a special function that evaluates to the following values:

- *`X`* when the “linked” outer expression *`oe_i`* is not `NULL`
- `TRUE` when the “linked” outer expression *`oe_i`* is `NULL`

Note

Trigger functions are *not* triggers of the kind that you create with `CREATE TRIGGER`.

Equalities that are wrapped within `trigcond()` functions are not first class predicates for the query optimizer. Most optimizations cannot deal with predicates that may be turned on and off at query execution time, so they assume any `trigcond(`*`X`*`)` to be an unknown function and ignore it. Triggered equalities can be used by those optimizations:

- Reference optimizations: `trigcond(`*`X`*`=`*`Y`*` [OR `*`Y`*` IS NULL])` can be used to construct `ref`, `eq_ref`, or `ref_or_null` table accesses.

- Index lookup-based subquery execution engines: `trigcond(X=Y)` can be used to construct `unique_subquery` or `index_subquery` accesses.
- Table-condition generator: If the subquery is a join of several tables, the triggered condition is checked as soon as possible.

When the optimizer uses a triggered condition to create some kind of index lookup-based access (as for the first two items of the preceding list), it must have a fallback strategy for the case when the condition is turned off. This fallback strategy is always the same: Do a full table scan. In `EXPLAIN` output, the fallback shows up as `Full scan on NULL key` in the `Extra` column:

```
mysql> EXPLAIN SELECT t1.col1,
       t1.col1 IN (SELECT t2.key1 FROM t2 WHERE t2.col2=t1.col2) FROM t1\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: t1
        ...
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: t2
         type: index_subquery
possible_keys: key1
          key: key1
      key_len: 5
          ref: func
         rows: 2
        Extra: Using where; Full scan on NULL key
```

If you run `EXPLAIN` followed by `SHOW WARNINGS`, you can see the triggered condition:

```
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: select `test`.`t1`.`col1` AS `col1`,
         <in_optimizer>(`test`.`t1`.`col1`,
         <exists>(<index_lookup>(<cache>(`test`.`t1`.`col1`) in t2
         on key1 checking NULL
         where (`test`.`t2`.`col2` = `test`.`t1`.`col2`) having
         trigcond(<is_not_null_test>(`test`.`t2`.`key1`))))) AS
         `t1.col1 IN (select t2.key1 from t2 where t2.col2=t1.col2)`
         from `test`.`t1`
```

The use of triggered conditions has some performance implications. A `NULL IN (SELECT ...)` expression now may cause a full table scan (which is slow) when it previously did not. This is the price paid for correct results (the goal of the trigger-condition strategy is to improve compliance, not speed).

For multiple-table subqueries, execution of `NULL IN (SELECT ...)` is particularly slow because the join optimizer does not optimize for the case where the outer expression is `NULL`. It assumes that subquery evaluations with `NULL` on the left side are very rare, even if there are statistics that indicate otherwise. On the other hand, if the outer expression might be `NULL` but never actually is, there is no performance penalty.

To help the query optimizer better execute your queries, use these suggestions:

- Declare a column as `NOT NULL` if it really is. This also helps other aspects of the optimizer by simplifying condition testing for the column.
- If you need not distinguish a `NULL` from `FALSE` subquery result, you can easily avoid the slow execution path. Replace a comparison that looks like this:

  ```
  outer_expr [NOT] IN (SELECT inner_expr FROM ...)
  ```

  with this expression:

  ```
  (outer_expr IS NOT NULL) AND (outer_expr [NOT] IN (SELECT inner_expr FROM ...))
  ```

Then `NULL IN (SELECT ...)` is never evaluated because MySQL stops evaluating `AND` parts as soon as the expression result is clear.

Another possible rewrite:

```
[NOT] EXISTS (SELECT inner_expr FROM ...
        WHERE inner_expr=outer_expr)
```

The `subquery_materialization_cost_based` flag of the `optimizer_switch` system variable enables control over the choice between subquery materialization and `IN`-to-`EXISTS` subquery transformation. See Section 8.9.2, “Switchable Optimizations”.

8.2.2.4 Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization

The optimizer can handle derived table references using two strategies (which also apply to view references and common table expressions):

- Merge the derived table into the outer query block
- Materialize the derived table to an internal temporary table

Example 1:

```
SELECT * FROM (SELECT * FROM t1) AS derived_t1;
```

With merging of the derived table `derived_t1`, that query is executed similar to:

```
SELECT * FROM t1;
```

Example 2:

```
SELECT *
  FROM t1 JOIN (SELECT t2.f1 FROM t2) AS derived_t2 ON t1.f2=derived_t2.f1
  WHERE t1.f1 > 0;
```

With merging of the derived table `derived_t2`, that query is executed similar to:

```
SELECT t1.*, t2.f1
  FROM t1 JOIN t2 ON t1.f2=t2.f1
  WHERE t1.f1 > 0;
```

With materialization, `derived_t1` and `derived_t2` are each treated as a separate table within their respective queries.

The optimizer handles derived tables, view references, and common table expressions the same way: It avoids unnecessary materialization whenever possible, which enables pushing down conditions from the outer query to derived tables and produces more efficient execution plans. (For an example, see Section 8.2.2.2, “Optimizing Subqueries with Materialization”.)

If merging would result in an outer query block that references more than 61 base tables, the optimizer chooses materialization instead.

The optimizer propagates an `ORDER BY` clause in a derived table or view reference to the outer query block if these conditions are all true:

- The outer query is not grouped or aggregated.
- The outer query does not specify `DISTINCT`, `HAVING`, or `ORDER BY`.
- The outer query has this derived table or view reference as the only source in the `FROM` clause.

Otherwise, the optimizer ignores the `ORDER BY` clause.

The following means are available to influence whether the optimizer attempts to merge derived tables, view references, and common table expressions into the outer query block:

- The `MERGE` and `NO_MERGE` optimizer hints can be used. They apply assuming that no other rule prevents merging. See Section 8.9.3, “Optimizer Hints”.
- Similarly, you can use the `derived_merge` flag of the `optimizer_switch` system variable. See Section 8.9.2, “Switchable Optimizations”. By default, the flag is enabled to permit merging. Disabling the flag prevents merging and avoids `ER_UPDATE_TABLE_USED` errors.

  The `derived_merge` flag also applies to views that contain no `ALGORITHM` clause. Thus, if an `ER_UPDATE_TABLE_USED` error occurs for a view reference that uses an expression equivalent to the subquery, adding `ALGORITHM=TEMPTABLE` to the view definition prevents merging and takes precedence over the `derived_merge` value.
- It is possible to disable merging by using in the subquery any constructs that prevent merging, although these are not as explicit in their effect on materialization. Constructs that prevent merging are the same for derived tables, common table expressions, and view references:
  - Aggregate functions or window functions (`SUM()`, `MIN()`, `MAX()`, `COUNT()`, and so forth)
  - `DISTINCT`
  - `GROUP BY`
  - `HAVING`
  - `LIMIT`
  - `UNION` or `UNION ALL`
  - Subqueries in the select list
  - Assignments to user variables
  - References only to literal values (in this case, there is no underlying table)

If the optimizer chooses the materialization strategy rather than merging for a derived table, it handles the query as follows:

- The optimizer postpones derived table materialization until its contents are needed during query execution. This improves performance because delaying materialization may result in not having to do it at all. Consider a query that joins the result of a derived table to another table: If the optimizer processes that other table first and finds that it returns no rows, the join need not be carried out further and the optimizer can completely skip materializing the derived table.
- During query execution, the optimizer may add an index to a derived table to speed up row retrieval from it.

Consider the following `EXPLAIN` statement, for a `SELECT` query that contains a derived table:

```
EXPLAIN SELECT * FROM (SELECT * FROM t1) AS derived_t1;
```

The optimizer avoids materializing the derived table by delaying it until the result is needed during `SELECT` execution. In this case, the query is not executed (because it occurs in an `EXPLAIN` statement), so the result is never needed.

Even for queries that are executed, delay of derived table materialization may enable the optimizer to avoid materialization entirely. When this happens, query execution is quicker by the time needed to perform materialization. Consider the following query, which joins the result of a derived table to another table:

```
SELECT *
  FROM t1 JOIN (SELECT t2.f1 FROM t2) AS derived_t2
          ON t1.f2=derived_t2.f1
  WHERE t1.f1 > 0;
```

If the optimization processes `t1` first and the `WHERE` clause produces an empty result, the join must necessarily be empty and the derived table need not be materialized.

For cases when a derived table requires materialization, the optimizer may add an index to the materialized table to speed up access to it. If such an index enables `ref` access to the table, it can greatly reduce amount of data read during query execution. Consider the following query:

```
SELECT *
 FROM t1 JOIN (SELECT DISTINCT f1 FROM t2) AS derived_t2
         ON t1.f1=derived_t2.f1;
```

The optimizer constructs an index over column `f1` from `derived_t2` if doing so would enable use of `ref` access for the lowest cost execution plan. After adding the index, the optimizer can treat the materialized derived table the same as a regular table with an index, and it benefits similarly from the generated index. The overhead of index creation is negligible compared to the cost of query execution without the index. If `ref` access would result in higher cost than some other access method, the optimizer creates no index and loses nothing.

For optimizer trace output, a merged derived table or view reference is not shown as a node. Only its underlying tables appear in the top query's plan.

What is true for materialization of derived tables is also true for common table expressions (CTEs). In addition, the following considerations pertain specifically to CTEs.

If a CTE is materialized by a query, it is materialized once for the query, even if the query references it several times.

A recursive CTE is always materialized.

If a CTE is materialized, the optimizer automatically adds relevant indexes if it estimates that indexing can speed up access by the top-level statement to the CTE. This is similar to automatic indexing of derived tables, except that if the CTE is referenced multiple times, the optimizer may create multiple indexes, to speed up access by each reference in the most appropriate way.

The `MERGE` and `NO_MERGE` optimizer hints can be applied to CTEs. Each CTE reference in the top-level statement can have its own hint, permitting CTE references to be selectively merged or materialized. The following statement uses hints to indicate that `cte1` should be merged and `cte2` should be materialized:

```
WITH
  cte1 AS (SELECT a, b FROM table1),
  cte2 AS (SELECT c, d FROM table2)
SELECT /*+ MERGE(cte1) NO_MERGE(cte2) */ cte1.b, cte2.d
FROM cte1 JOIN cte2
WHERE cte1.a = cte2.c;
```

The `ALGORITHM` clause for `CREATE VIEW` does not affect materialization for any `WITH` clause preceding the `SELECT` statement in the view definition. Consider this statement:

```
CREATE ALGORITHM={TEMPTABLE|MERGE} VIEW v1 AS WITH ... SELECT ...
```

The `ALGORITHM` value affects materialization only of the `SELECT`, not the `WITH` clause.

Prior to MySQL 8.0.16, if `internal_tmp_disk_storage_engine=MYISAM`, an error occurred for any attempt to materialize a CTE using an on-disk temporary table, since for CTEs, the storage engine used for on-disk internal temporary tables could not be `MyISAM`. Beginning with MySQL 8.0.16, this is no longer an issue, since `TempTable` now always uses `InnoDB` for on-disk internal temporary tables.

As mentioned previously, a CTE, if materialized, is materialized once, even if referenced multiple times. To indicate one-time materialization, optimizer trace output contains an occurrence of `creating_tmp_table` plus one or more occurrences of `reusing_tmp_table`.

CTEs are similar to derived tables, for which the `materialized_from_subquery` node follows the reference. This is true for a CTE that is referenced multiple times, so there is no duplication of `materialized_from_subquery` nodes (which would give the impression that the subquery is executed multiple times, and produce unnecessarily verbose output). Only one reference to the CTE has a complete `materialized_from_subquery` node with the description of its subquery plan. Other references have a reduced `materialized_from_subquery` node. The same idea applies to `EXPLAIN` output in `TRADITIONAL` format: Subqueries for other references are not shown.

8.2.2.5 Derived Condition Pushdown Optimization

MySQL 8.0.22 and later supports derived condition pushdown for eligible subqueries. For a query such as `SELECT * FROM (SELECT i, j FROM t1) AS dt WHERE i > constant`, it is possible in many cases to push the outer `WHERE` condition down to the derived table, in this case resulting in `SELECT * FROM (SELECT i, j FROM t1 WHERE i > constant) AS dt`. When a derived table cannot be merged into the outer query (for example, if the derived table uses aggregation), pushing the outer `WHERE` condition down to the derived table should decrease the number of rows that need to be processed and thus speed up execution of the query.

Note

Prior to MySQL 8.0.22, if a derived table was materialized but not merged, MySQL materialized the entire table, then qualified all of the resulting rows with the `WHERE` condition. This is still the case if derived condition pushdown is not enabled, or cannot be employed for some other reason.

Outer `WHERE` conditions can be pushed down to derived materialized tables under the following circumstances:

- When the derived table uses no aggregate or window functions, the outer `WHERE` condition can be pushed down to it directly. This includes `WHERE` conditions having multiple predicates joined with `AND`, `OR`, or both.

  For example, the query `SELECT * FROM (SELECT f1, f2 FROM t1) AS dt WHERE f1 < 3 AND f2 > 11` is rewritten as `SELECT f1, f2 FROM (SELECT f1, f2 FROM t1 WHERE f1 < 3 AND f2 > 11) AS dt`.

- When the derived table has a `GROUP BY` and uses no window functions, an outer `WHERE` condition referencing one or more columns which are not part of the `GROUP BY` can be pushed down to the derived table as a `HAVING` condition.

  For example, `SELECT * FROM (SELECT i, j, SUM(k) AS sum FROM t1 GROUP BY i, j) AS dt WHERE sum > 100` is rewritten following derived condition pushdown as `SELECT * FROM (SELECT i, j, SUM(k) AS sum FROM t1 GROUP BY i, j HAVING sum > 100) AS dt`.

- When the derived table uses a `GROUP BY` and the columns in the outer `WHERE` condition are `GROUP BY` columns, the `WHERE` conditions referencing those columns can be pushed down directly to the derived table.

  For example, the query `SELECT * FROM (SELECT i,j, SUM(k) AS sum FROM t1 GROUP BY i,j) AS dt WHERE i > 10` is rewritten as `SELECT * FROM (SELECT i,j, SUM(k) AS sum FROM t1 WHERE i > 10 GROUP BY i,j) AS dt`.

  In the event that the outer `WHERE` condition has predicates referencing columns which are part of the `GROUP BY` as well as predicates referencing columns which are not, predicates of the former sort are pushed down as `WHERE` conditions, while those of the latter type are pushed down as `HAVING` conditions. For example, in the query `SELECT * FROM (SELECT i, j, SUM(k) AS sum FROM`

`t1 GROUP BY i,j) AS dt WHERE i > 10 AND sum > 100`, the predicate `i > 10` in the outer `WHERE` clause references a `GROUP BY` column, whereas the predicate `sum > 100` does not reference any `GROUP BY` column. Thus the derived table pushdown optimization causes the query to be rewritten in a manner similar to what is shown here:

```
SELECT * FROM (
    SELECT i, j, SUM(k) AS sum FROM t1
        WHERE i > 10
        GROUP BY i, j
        HAVING sum > 100
    ) AS dt;
```

To enable derived condition pushdown, the `optimizer_switch` system variable's `derived_condition_pushdown` flag (added in this release) must be set to `on`, which is the default setting. If this optimization is disabled by `optimizer_switch`, you can enable it for a specific query using the `DERIVED_CONDITION_PUSHDOWN` optimizer hint. To disable the optimization for a given query, use the `NO_DERIVED_CONDITION_PUSHDOWN` optimizer hint.

The following restrictions and limitations apply to the derived table condition pushdown optimization:

- The optimization cannot be used if the derived table contains `UNION`. This restriction is lifted in MySQL 8.0.29. Consider two tables `t1` and `t2`, and a view `v` containing their union, created as shown here:

  ```
  CREATE TABLE t1 (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    c1 INT,
    KEY i1 (c1)
  );

  CREATE TABLE t2 (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    c1 INT,
    KEY i1 (c1)
  );

  CREATE OR REPLACE VIEW v AS
       SELECT id, c1 FROM t1
       UNION ALL
       SELECT id, c1 FROM t2;
  ```

  As be seen in the output of `EXPLAIN`, a condition present in the top level of a query such as `SELECT * FROM v WHERE c1 = 12` can now be pushed down to both query blocks in the derived table:

  ```
  mysql> EXPLAIN FORMAT=TREE SELECT * FROM v WHERE c1 = 12\G
  *************************** 1. row ***************************
  EXPLAIN: -> Table scan on v  (cost=1.26..2.52 rows=2)
      -> Union materialize  (cost=2.16..3.42 rows=2)
          -> Covering index lookup on t1 using i1 (c1=12)  (cost=0.35 rows=1)
          -> Covering index lookup on t2 using i1 (c1=12)  (cost=0.35 rows=1)

  1 row in set (0.00 sec)
  ```

  In MySQL 8.0.29 and later, the derived table condition pushdown optimization can be employed with `UNION` queries, with the following exceptions:

  - Condition pushdown cannot be used with a `UNION` query if any materialized derived table that is part of the `UNION` is a recursive common table expression (see Recursive Common Table Expressions).
  - Conditions containing nondeterministic expressions cannot be pushed down to a derived table.

- The derived table cannot use a `LIMIT` clause.
- Conditions containing subqueries cannot be pushed down.
- The optimization cannot be used if the derived table is an inner table of an outer join.

- If a materialized derived table is a common table expression, conditions are not pushed down to it if it is referenced multiple times.
- Conditions using parameters can be pushed down if the condition is of the form *`derived_column > ?`*. If a derived column in an outer `WHERE` condition is an expression having a `?` in the underlying derived table, this condition cannot be pushed down.
- For a query in which the condition is on the tables of a view created using `ALGORITHM=TEMPTABLE` instead of on the view itself, the multiple equality is not recognized at resolution, and thus the condition cannot be not pushed down. This because, when optimizing a query, condition pushdown takes place during resolution phase while multiple equality propagation occurs during optimization.

  This is not an issue in such cases for a view using `ALGORITHM=MERGE`, where the equality can be propagated and the condition pushed down.
- Beginning with MySQL 8.0.28, a condition cannot be pushed down if the derived table's `SELECT` list contain any assignments to user variables. (Bug #104918)

8.2.3 Optimizing INFORMATION_SCHEMA Queries

Applications that monitor databases may make frequent use of `INFORMATION_SCHEMA` tables. To write queries for these tables most efficiently, use the following general guidelines:

- Try to query only `INFORMATION_SCHEMA` tables that are views on data dictionary tables.
- Try to query only for static metadata. Selecting columns or using retrieval conditions for dynamic metadata along with static metadata adds overhead to process the dynamic metadata.

Note

Comparison behavior for database and table names in `INFORMATION_SCHEMA` queries might differ from what you expect. For details, see Section 10.8.7, “Using Collation in INFORMATION_SCHEMA Searches”.

These `INFORMATION_SCHEMA` tables are implemented as views on data dictionary tables, so queries on them retrieve information from the data dictionary:

```
CHARACTER_SETS
CHECK_CONSTRAINTS
COLLATIONS
COLLATION_CHARACTER_SET_APPLICABILITY
COLUMNS
EVENTS
FILES
INNODB_COLUMNS
INNODB_DATAFILES
INNODB_FIELDS
INNODB_FOREIGN
INNODB_FOREIGN_COLS
INNODB_INDEXES
INNODB_TABLES
INNODB_TABLESPACES
INNODB_TABLESPACES_BRIEF
INNODB_TABLESTATS
KEY_COLUMN_USAGE
PARAMETERS
PARTITIONS
REFERENTIAL_CONSTRAINTS
RESOURCE_GROUPS
ROUTINES
SCHEMATA
STATISTICS
TABLES
TABLE_CONSTRAINTS
TRIGGERS
VIEWS
```

```
VIEW_ROUTINE_USAGE
VIEW_TABLE_USAGE
```

Some types of values, even for a non-view `INFORMATION_SCHEMA` table, are retrieved by lookups from the data dictionary. This includes values such as database and table names, table types, and storage engines.

Some `INFORMATION_SCHEMA` tables contain columns that provide table statistics:

```
STATISTICS.CARDINALITY
TABLES.AUTO_INCREMENT
TABLES.AVG_ROW_LENGTH
TABLES.CHECKSUM
TABLES.CHECK_TIME
TABLES.CREATE_TIME
TABLES.DATA_FREE
TABLES.DATA_LENGTH
TABLES.INDEX_LENGTH
TABLES.MAX_DATA_LENGTH
TABLES.TABLE_ROWS
TABLES.UPDATE_TIME
```

Those columns represent dynamic table metadata; that is, information that changes as table contents change.

By default, MySQL retrieves cached values for those columns from the `mysql.index_stats` and `mysql.table_stats` dictionary tables when the columns are queried, which is more efficient than retrieving statistics directly from the storage engine. If cached statistics are not available or have expired, MySQL retrieves the latest statistics from the storage engine and caches them in the `mysql.index_stats` and `mysql.table_stats` dictionary tables. Subsequent queries retrieve the cached statistics until the cached statistics expire. A server restart or the first opening of the `mysql.index_stats` and `mysql.table_stats` tables do not update cached statistics automatically.

The `information_schema_stats_expiry` session variable defines the period of time before cached statistics expire. The default is 86400 seconds (24 hours), but the time period can be extended to as much as one year.

To update cached values at any time for a given table, use `ANALYZE TABLE`.

Querying statistics columns does not store or update statistics in the `mysql.index_stats` and `mysql.table_stats` dictionary tables under these circumstances:

- When cached statistics have not expired.
- When `information_schema_stats_expiry` is set to 0.
- When the server is in `read_only`, `super_read_only`, `transaction_read_only`, or `innodb_read_only` mode.
- When the query also fetches Performance Schema data.

`information_schema_stats_expiry` is a session variable, and each client session can define its own expiration value. Statistics that are retrieved from the storage engine and cached by one session are available to other sessions.

Note

If the `innodb_read_only` system variable is enabled, `ANALYZE TABLE` may fail because it cannot update statistics tables in the data dictionary, which use `InnoDB`. For `ANALYZE TABLE` operations that update the key distribution, failure may occur even if the operation updates the table itself (for example, if it is a `MyISAM` table). To obtain the updated distribution statistics, set `information_schema_stats_expiry=0`.

For INFORMATION_SCHEMA tables implemented as views on data dictionary tables, indexes on the underlying data dictionary tables permit the optimizer to construct efficient query execution plans. To see the choices made by the optimizer, use EXPLAIN. To also see the query used by the server to execute an INFORMATION_SCHEMA query, use SHOW WARNINGS immediately following EXPLAIN.

Consider this statement, which identifies collations for the utf8mb4 character set:

```
mysql> SELECT COLLATION_NAME
       FROM INFORMATION_SCHEMA.COLLATION_CHARACTER_SET_APPLICABILITY
       WHERE CHARACTER_SET_NAME = 'utf8mb4';
+----------------------------+
| COLLATION_NAME             |
+----------------------------+
| utf8mb4_general_ci         |
| utf8mb4_bin                |
| utf8mb4_unicode_ci         |
| utf8mb4_icelandic_ci       |
| utf8mb4_latvian_ci         |
| utf8mb4_romanian_ci        |
| utf8mb4_slovenian_ci       |
...
```

How does the server process that statement? To find out, use EXPLAIN:

```
mysql> EXPLAIN SELECT COLLATION_NAME
       FROM INFORMATION_SCHEMA.COLLATION_CHARACTER_SET_APPLICABILITY
       WHERE CHARACTER_SET_NAME = 'utf8mb4'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: cs
   partitions: NULL
         type: const
possible_keys: PRIMARY,name
          key: name
      key_len: 194
          ref: const
         rows: 1
     filtered: 100.00
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: col
   partitions: NULL
         type: ref
possible_keys: character_set_id
          key: character_set_id
      key_len: 8
          ref: const
         rows: 68
     filtered: 100.00
        Extra: NULL
2 rows in set, 1 warning (0.01 sec)
```

To see the query used to satisfy that statement, use SHOW WARNINGS:

```
mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `mysql`.`col`.`name` AS `COLLATION_NAME`
         from `mysql`.`character_sets` `cs`
         join `mysql`.`collations` `col`
         where ((`mysql`.`col`.`character_set_id` = '45')
         and ('utf8mb4' = 'utf8mb4'))
```

As indicated by SHOW WARNINGS, the server handles the query on COLLATION_CHARACTER_SET_APPLICABILITY as a query on the character_sets and collations data dictionary tables in the mysql system database.

8.2.4 Optimizing Performance Schema Queries

Applications that monitor databases may make frequent use of Performance Schema tables. To write queries for these tables most efficiently, take advantage of their indexes. For example, include a `WHERE` clause that restricts retrieved rows based on comparison to specific values in an indexed column.

Most Performance Schema tables have indexes. Tables that do not are those that normally contain few rows or are unlikely to be queried frequently. Performance Schema indexes give the optimizer access to execution plans other than full table scans. These indexes also improve performance for related objects, such as `sys` schema views that use those tables.

To see whether a given Performance Schema table has indexes and what they are, use `SHOW INDEX` or `SHOW CREATE TABLE`:

```
mysql> SHOW INDEX FROM performance_schema.accounts\G
*************************** 1. row ***************************
        Table: accounts
   Non_unique: 0
     Key_name: ACCOUNT
 Seq_in_index: 1
  Column_name: USER
    Collation: NULL
  Cardinality: NULL
     Sub_part: NULL
       Packed: NULL
         Null: YES
   Index_type: HASH
      Comment:
Index_comment:
      Visible: YES
*************************** 2. row ***************************
        Table: accounts
   Non_unique: 0
     Key_name: ACCOUNT
 Seq_in_index: 2
  Column_name: HOST
    Collation: NULL
  Cardinality: NULL
     Sub_part: NULL
       Packed: NULL
         Null: YES
   Index_type: HASH
      Comment:
Index_comment:
      Visible: YES

mysql> SHOW CREATE TABLE performance_schema.rwlock_instances\G
*************************** 1. row ***************************
       Table: rwlock_instances
Create Table: CREATE TABLE `rwlock_instances` (
  `NAME` varchar(128) NOT NULL,
  `OBJECT_INSTANCE_BEGIN` bigint(20) unsigned NOT NULL,
  `WRITE_LOCKED_BY_THREAD_ID` bigint(20) unsigned DEFAULT NULL,
  `READ_LOCKED_BY_COUNT` int(10) unsigned NOT NULL,
  PRIMARY KEY (`OBJECT_INSTANCE_BEGIN`),
  KEY `NAME` (`NAME`),
  KEY `WRITE_LOCKED_BY_THREAD_ID` (`WRITE_LOCKED_BY_THREAD_ID`)
) ENGINE=PERFORMANCE_SCHEMA DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

To see the execution plan for a Performance Schema query and whether it uses any indexes, use `EXPLAIN`:

```
mysql> EXPLAIN SELECT * FROM performance_schema.accounts
       WHERE (USER,HOST) = ('root','localhost')\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: accounts
   partitions: NULL
```

```
         type: const
possible_keys: ACCOUNT
          key: ACCOUNT
      key_len: 278
          ref: const,const
         rows: 1
     filtered: 100.00
        Extra: NULL
```

The `EXPLAIN` output indicates that the optimizer uses the `accounts` table `ACCOUNT` index that comprises the `USER` and `HOST` columns.

Performance Schema indexes are virtual: They are a construct of the Performance Schema storage engine and use no memory or disk storage. The Performance Schema reports index information to the optimizer so that it can construct efficient execution plans. The Performance Schema in turn uses optimizer information about what to look for (for example, a particular key value), so that it can perform efficient lookups without building actual index structures. This implementation provides two important benefits:

- It entirely avoids the maintenance cost normally incurred for tables that undergo frequent updates.
- It reduces at an early stage of query execution the amount of data retrieved. For conditions on the indexed columns, the Performance Schema efficiently returns only table rows that satisfy the query conditions. Without an index, the Performance Schema would return all rows in the table, requiring that the optimizer later evaluate the conditions against each row to produce the final result.

Performance Schema indexes are predefined and cannot be dropped, added, or altered.

Performance Schema indexes are similar to hash indexes. For example:

- They are used only for equality comparisons that use the `=` or `<=>` operators.
- They are unordered. If a query result must have specific row ordering characteristics, include an `ORDER BY` clause.

For additional information about hash indexes, see Section 8.3.9, “Comparison of B-Tree and Hash Indexes”.

8.2.5 Optimizing Data Change Statements

This section explains how to speed up data change statements: `INSERT`, `UPDATE`, and `DELETE`. Traditional OLTP applications and modern web applications typically do many small data change operations, where concurrency is vital. Data analysis and reporting applications typically run data change operations that affect many rows at once, where the main considerations is the I/O to write large amounts of data and keep indexes up-to-date. For inserting and updating large volumes of data (known in the industry as ETL, for “extract-transform-load”), sometimes you use other SQL statements or external commands, that mimic the effects of `INSERT`, `UPDATE`, and `DELETE` statements.

8.2.5.1 Optimizing INSERT Statements

To optimize insert speed, combine many small operations into a single large operation. Ideally, you make a single connection, send the data for many new rows at once, and delay all index updates and consistency checking until the very end.

The time required for inserting a row is determined by the following factors, where the numbers indicate approximate proportions:

- Connecting: (3)
- Sending query to server: (2)
- Parsing query: (2)

- Inserting row: (1 × size of row)
- Inserting indexes: (1 × number of indexes)
- Closing: (1)

This does not take into consideration the initial overhead to open tables, which is done once for each concurrently running query.

The size of the table slows down the insertion of indexes by log *N*, assuming B-tree indexes.

You can use the following methods to speed up inserts:

- If you are inserting many rows from the same client at the same time, use `INSERT` statements with multiple `VALUES` lists to insert several rows at a time. This is considerably faster (many times faster in some cases) than using separate single-row `INSERT` statements. If you are adding data to a nonempty table, you can tune the `bulk_insert_buffer_size` variable to make data insertion even faster. See Section 5.1.8, “Server System Variables”.
- When loading a table from a text file, use `LOAD DATA`. This is usually 20 times faster than using `INSERT` statements. See Section 13.2.9, “LOAD DATA Statement”.
- Take advantage of the fact that columns have default values. Insert values explicitly only when the value to be inserted differs from the default. This reduces the parsing that MySQL must do and improves the insert speed.
- See Section 8.5.5, “Bulk Data Loading for InnoDB Tables” for tips specific to `InnoDB` tables.
- See Section 8.6.2, “Bulk Data Loading for MyISAM Tables” for tips specific to `MyISAM` tables.

8.2.5.2 Optimizing UPDATE Statements

An update statement is optimized like a `SELECT` query with the additional overhead of a write. The speed of the write depends on the amount of data being updated and the number of indexes that are updated. Indexes that are not changed do not get updated.

Another way to get fast updates is to delay updates and then do many updates in a row later. Performing multiple updates together is much quicker than doing one at a time if you lock the table.

For a `MyISAM` table that uses dynamic row format, updating a row to a longer total length may split the row. If you do this often, it is very important to use `OPTIMIZE TABLE` occasionally. See Section 13.7.3.4, “OPTIMIZE TABLE Statement”.

8.2.5.3 Optimizing DELETE Statements

The time required to delete individual rows in a `MyISAM` table is exactly proportional to the number of indexes. To delete rows more quickly, you can increase the size of the key cache by increasing the `key_buffer_size` system variable. See Section 5.1.1, “Configuring the Server”.

To delete all rows from a `MyISAM` table, `TRUNCATE TABLE *tbl_name*` is faster than `DELETE FROM *tbl_name*`. Truncate operations are not transaction-safe; an error occurs when attempting one in the course of an active transaction or active table lock. See Section 13.1.37, “TRUNCATE TABLE Statement”.

8.2.6 Optimizing Database Privileges

The more complex your privilege setup, the more overhead applies to all SQL statements. Simplifying the privileges established by `GRANT` statements enables MySQL to reduce permission-checking overhead when clients execute statements. For example, if you do not grant any table-level or column-level privileges, the server need not ever check the contents of the `tables_priv` and `columns_priv` tables. Similarly, if you place no resource limits on any accounts, the server does not have to perform

resource counting. If you have a very high statement-processing load, consider using a simplified grant structure to reduce permission-checking overhead.

8.2.7 Other Optimization Tips

This section lists a number of miscellaneous tips for improving query processing speed:

- If your application makes several database requests to perform related updates, combining the statements into a stored routine can help performance. Similarly, if your application computes a single result based on several column values or large volumes of data, combining the computation into a loadable function can help performance. The resulting fast database operations are then available to be reused by other queries, applications, and even code written in different programming languages. See Section 25.2, “Using Stored Routines” and Adding Functions to MySQL for more information.
- To fix any compression issues that occur with `ARCHIVE` tables, use `OPTIMIZE TABLE`. See Section 16.5, “The ARCHIVE Storage Engine”.
- If possible, classify reports as “live” or as “statistical”, where data needed for statistical reports is created only from summary tables that are generated periodically from the live data.
- If you have data that does not conform well to a rows-and-columns table structure, you can pack and store data into a `BLOB` column. In this case, you must provide code in your application to pack and unpack information, but this might save I/O operations to read and write the sets of related values.
- With Web servers, store images and other binary assets as files, with the path name stored in the database rather than the file itself. Most Web servers are better at caching files than database contents, so using files is generally faster. (Although you must handle backups and storage issues yourself in this case.)
- If you need really high speed, look at the low-level MySQL interfaces. For example, by accessing the MySQL `InnoDB` or `MyISAM` storage engine directly, you could get a substantial speed increase compared to using the SQL interface.

  Similarly, for databases using the `NDBCLUSTER` storage engine, you may wish to investigate possible use of the NDB API (see MySQL NDB Cluster API Developer Guide).
- Replication can provide a performance benefit for some operations. You can distribute client retrievals among replicas to split up the load. To avoid slowing down the source while making backups, you can make backups using a replica. See Chapter 17, *Replication*.

8.3 Optimization and Indexes

The best way to improve the performance of `SELECT` operations is to create indexes on one or more of the columns that are tested in the query. The index entries act like pointers to the table rows, allowing the query to quickly determine which rows match a condition in the `WHERE` clause, and retrieve the other column values for those rows. All MySQL data types can be indexed.

Although it can be tempting to create an indexes for every possible column used in a query, unnecessary indexes waste space and waste time for MySQL to determine which indexes to use. Indexes also add to the cost of inserts, updates, and deletes because each index must be updated. You must find the right balance to achieve fast queries using the optimal set of indexes.

8.3.1 How MySQL Uses Indexes

Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. The larger the table, the more this costs. If the table has an index for the columns in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data. This is much faster than reading every row sequentially.

Most MySQL indexes (`PRIMARY KEY`, `UNIQUE`, `INDEX`, and `FULLTEXT`) are stored in B-trees. Exceptions: Indexes on spatial data types use R-trees; `MEMORY` tables also support hash indexes; `InnoDB` uses inverted lists for `FULLTEXT` indexes.

In general, indexes are used as described in the following discussion. Characteristics specific to hash indexes (as used in `MEMORY` tables) are described in Section 8.3.9, “Comparison of B-Tree and Hash Indexes”.

MySQL uses indexes for these operations:

- To find the rows matching a `WHERE` clause quickly.
- To eliminate rows from consideration. If there is a choice between multiple indexes, MySQL normally uses the index that finds the smallest number of rows (the most selective index).
- If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to look up rows. For example, if you have a three-column index on `(col1, col2, col3)`, you have indexed search capabilities on `(col1)`, `(col1, col2)`, and `(col1, col2, col3)`. For more information, see Section 8.3.6, “Multiple-Column Indexes”.
- To retrieve rows from other tables when performing joins. MySQL can use indexes on columns more efficiently if they are declared as the same type and size. In this context, `VARCHAR` and `CHAR` are considered the same if they are declared as the same size. For example, `VARCHAR(10)` and `CHAR(10)` are the same size, but `VARCHAR(10)` and `CHAR(15)` are not.

  For comparisons between nonbinary string columns, both columns should use the same character set. For example, comparing a `utf8mb4` column with a `latin1` column precludes use of an index.

  Comparison of dissimilar columns (comparing a string column to a temporal or numeric column, for example) may prevent use of indexes if values cannot be compared directly without conversion. For a given value such as `1` in the numeric column, it might compare equal to any number of values in the string column such as `'1'`, `' 1'`, `'00001'`, or `'01.e1'`. This rules out use of any indexes for the string column.
- To find the `MIN()` or `MAX()` value for a specific indexed column *`key_col`*. This is optimized by a preprocessor that checks whether you are using `WHERE` *`key_part_N = constant`* on all key parts that occur before *`key_col`* in the index. In this case, MySQL does a single key lookup for each `MIN()` or `MAX()` expression and replaces it with a constant. If all expressions are replaced with constants, the query returns at once. For example:

```
SELECT MIN(key_part2),MAX(key_part2)
  FROM tbl_name WHERE key_part1=10;
```

- To sort or group a table if the sorting or grouping is done on a leftmost prefix of a usable index (for example, `ORDER BY` *`key_part1, key_part2`*). If all key parts are followed by `DESC`, the key is read in reverse order. (Or, if the index is a descending index, the key is read in forward order.) See Section 8.2.1.16, “ORDER BY Optimization”, Section 8.2.1.17, “GROUP BY Optimization”, and Section 8.3.13, “Descending Indexes”.
- In some cases, a query can be optimized to retrieve values without consulting the data rows. (An index that provides all the necessary results for a query is called a covering index.) If a query uses from a table only columns that are included in some index, the selected values can be retrieved from the index tree for greater speed:

```
SELECT key_part3 FROM tbl_name
  WHERE key_part1=1
```

Indexes are less important for queries on small tables, or big tables where report queries process most or all of the rows. When a query needs to access most of the rows, reading sequentially is faster than working through an index. Sequential reads minimize disk seeks, even if not all the rows are needed for the query. See Section 8.2.1.23, “Avoiding Full Table Scans” for details.

8.3.2 Primary Key Optimization

The primary key for a table represents the column or set of columns that you use in your most vital queries. It has an associated index, for fast query performance. Query performance benefits from the `NOT NULL` optimization, because it cannot include any `NULL` values. With the `InnoDB` storage engine, the table data is physically organized to do ultra-fast lookups and sorts based on the primary key column or columns.

If your table is big and important, but does not have an obvious column or set of columns to use as a primary key, you might create a separate column with auto-increment values to use as the primary key. These unique IDs can serve as pointers to corresponding rows in other tables when you join tables using foreign keys.

8.3.3 SPATIAL Index Optimization

MySQL permits creation of `SPATIAL` indexes on `NOT NULL` geometry-valued columns (see Section 11.4.10, “Creating Spatial Indexes”). The optimizer checks the `SRID` attribute for indexed columns to determine which spatial reference system (SRS) to use for comparisons, and uses calculations appropriate to the SRS. (Prior to MySQL 8.0, the optimizer performs comparisons of `SPATIAL` index values using Cartesian calculations; the results of such operations are undefined if the column contains values with non-Cartesian SRIDs.)

For comparisons to work properly, each column in a `SPATIAL` index must be SRID-restricted. That is, the column definition must include an explicit `SRID` attribute, and all column values must have the same SRID.

The optimizer considers `SPATIAL` indexes only for SRID-restricted columns:

- Indexes on columns restricted to a Cartesian SRID enable Cartesian bounding box computations.
- Indexes on columns restricted to a geographic SRID enable geographic bounding box computations.

The optimizer ignores `SPATIAL` indexes on columns that have no `SRID` attribute (and thus are not SRID-restricted). MySQL still maintains such indexes, as follows:

- They are updated for table modifications (`INSERT`, `UPDATE`, `DELETE`, and so forth). Updates occur as though the index was Cartesian, even though the column might contain a mix of Cartesian and geographical values.
- They exist only for backward compatibility (for example, the ability to perform a dump in MySQL 5.7 and restore in MySQL 8.0). Because `SPATIAL` indexes on columns that are not SRID-restricted are of no use to the optimizer, each such column should be modified:
  - Verify that all values within the column have the same SRID. To determine the SRIDs contained in a geometry column *`col_name`*, use the following query:

    ```
    SELECT DISTINCT ST_SRID(col_name) FROM tbl_name;
    ```

    If the query returns more than one row, the column contains a mix of SRIDs. In that case, modify its contents so all values have the same SRID.
  - Redefine the column to have an explicit `SRID` attribute.
  - Recreate the `SPATIAL` index.

8.3.4 Foreign Key Optimization

If a table has many columns, and you query many different combinations of columns, it might be efficient to split the less-frequently used data into separate tables with a few columns each, and relate them back to the main table by duplicating the numeric ID column from the main table. That way,

each small table can have a primary key for fast lookups of its data, and you can query just the set of columns that you need using a join operation. Depending on how the data is distributed, the queries might perform less I/O and take up less cache memory because the relevant columns are packed together on disk. (To maximize performance, queries try to read as few data blocks as possible from disk; tables with only a few columns can fit more rows in each data block.)

8.3.5 Column Indexes

The most common type of index involves a single column, storing copies of the values from that column in a data structure, allowing fast lookups for the rows with the corresponding column values. The B-tree data structure lets the index quickly find a specific value, a set of values, or a range of values, corresponding to operators such as `=`, `>`, `≤`, `BETWEEN`, `IN`, and so on, in a `WHERE` clause.

The maximum number of indexes per table and the maximum index length is defined per storage engine. See Chapter 15, *The InnoDB Storage Engine*, and Chapter 16, *Alternative Storage Engines*. All storage engines support at least 16 indexes per table and a total index length of at least 256 bytes. Most storage engines have higher limits.

For additional information about column indexes, see Section 13.1.15, “CREATE INDEX Statement”.

- Index Prefixes
- FULLTEXT Indexes
- Spatial Indexes
- Indexes in the MEMORY Storage Engine

Index Prefixes

With *`col_name(N)`* syntax in an index specification for a string column, you can create an index that uses only the first *`N`* characters of the column. Indexing only a prefix of column values in this way can make the index file much smaller. When you index a `BLOB` or `TEXT` column, you *must* specify a prefix length for the index. For example:

```
CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));
```

Prefixes can be up to 767 bytes long for `InnoDB` tables that use the `REDUNDANT` or `COMPACT` row format. The prefix length limit is 3072 bytes for `InnoDB` tables that use the `DYNAMIC` or `COMPRESSED` row format. For MyISAM tables, the prefix length limit is 1000 bytes.

Note

Prefix limits are measured in bytes, whereas the prefix length in `CREATE TABLE`, `ALTER TABLE`, and `CREATE INDEX` statements is interpreted as number of characters for nonbinary string types (`CHAR`, `VARCHAR`, `TEXT`) and number of bytes for binary string types (`BINARY`, `VARBINARY`, `BLOB`). Take this into account when specifying a prefix length for a nonbinary string column that uses a multibyte character set.

If a search term exceeds the index prefix length, the index is used to exclude non-matching rows, and the remaining rows are examined for possible matches.

For additional information about index prefixes, see Section 13.1.15, “CREATE INDEX Statement”.

FULLTEXT Indexes

`FULLTEXT` indexes are used for full-text searches. Only the `InnoDB` and `MyISAM` storage engines support `FULLTEXT` indexes and only for `CHAR`, `VARCHAR`, and `TEXT` columns. Indexing always takes place over the entire column and column prefix indexing is not supported. For details, see Section 12.10, “Full-Text Search Functions”.

Optimizations are applied to certain kinds of `FULLTEXT` queries against single `InnoDB` tables. Queries with these characteristics are particularly efficient:

- `FULLTEXT` queries that only return the document ID, or the document ID and the search rank.
- `FULLTEXT` queries that sort the matching rows in descending order of score and apply a `LIMIT` clause to take the top N matching rows. For this optimization to apply, there must be no `WHERE` clauses and only a single `ORDER BY` clause in descending order.
- `FULLTEXT` queries that retrieve only the `COUNT(*)` value of rows matching a search term, with no additional `WHERE` clauses. Code the `WHERE` clause as `WHERE MATCH(text) AGAINST ('other_text')`, without any `> 0` comparison operator.

For queries that contain full-text expressions, MySQL evaluates those expressions during the optimization phase of query execution. The optimizer does not just look at full-text expressions and make estimates, it actually evaluates them in the process of developing an execution plan.

An implication of this behavior is that `EXPLAIN` for full-text queries is typically slower than for non-full-text queries for which no expression evaluation occurs during the optimization phase.

`EXPLAIN` for full-text queries may show `Select tables optimized away` in the `Extra` column due to matching occurring during optimization; in this case, no table access need occur during later execution.

Spatial Indexes

You can create indexes on spatial data types. `MyISAM` and `InnoDB` support R-tree indexes on spatial types. Other storage engines use B-trees for indexing spatial types (except for `ARCHIVE`, which does not support spatial type indexing).

Indexes in the MEMORY Storage Engine

The `MEMORY` storage engine uses `HASH` indexes by default, but also supports `BTREE` indexes.

8.3.6 Multiple-Column Indexes

MySQL can create composite indexes (that is, indexes on multiple columns). An index may consist of up to 16 columns. For certain data types, you can index a prefix of the column (see Section 8.3.5, “Column Indexes”).

MySQL can use multiple-column indexes for queries that test all the columns in the index, or queries that test just the first column, the first two columns, the first three columns, and so on. If you specify the columns in the right order in the index definition, a single composite index can speed up several kinds of queries on the same table.

A multiple-column index can be considered a sorted array, the rows of which contain values that are created by concatenating the values of the indexed columns.

Note

As an alternative to a composite index, you can introduce a column that is “hashed” based on information from other columns. If this column is short, reasonably unique, and indexed, it might be faster than a “wide” index on many columns. In MySQL, it is very easy to use this extra column:

```
SELECT * FROM tbl_name
  WHERE hash_col=MD5(CONCAT(val1,val2))
  AND col1=val1 AND col2=val2;
```

Suppose that a table has the following specification:

```
CREATE TABLE test (
    id         INT NOT NULL,
    last_name  CHAR(30) NOT NULL,
    first_name CHAR(30) NOT NULL,
    PRIMARY KEY (id),
    INDEX name (last_name,first_name)
);
```

The `name` index is an index over the `last_name` and `first_name` columns. The index can be used for lookups in queries that specify values in a known range for combinations of `last_name` and `first_name` values. It can also be used for queries that specify just a `last_name` value because that column is a leftmost prefix of the index (as described later in this section). Therefore, the `name` index is used for lookups in the following queries:

```
SELECT * FROM test WHERE last_name='Jones';

SELECT * FROM test
  WHERE last_name='Jones' AND first_name='John';

SELECT * FROM test
  WHERE last_name='Jones'
  AND (first_name='John' OR first_name='Jon');

SELECT * FROM test
  WHERE last_name='Jones'
  AND first_name >='M' AND first_name < 'N';
```

However, the `name` index is *not* used for lookups in the following queries:

```
SELECT * FROM test WHERE first_name='John';

SELECT * FROM test
  WHERE last_name='Jones' OR first_name='John';
```

Suppose that you issue the following `SELECT` statement:

```
SELECT * FROM tbl_name
  WHERE col1=val1 AND col2=val2;
```

If a multiple-column index exists on `col1` and `col2`, the appropriate rows can be fetched directly. If separate single-column indexes exist on `col1` and `col2`, the optimizer attempts to use the Index Merge optimization (see Section 8.2.1.3, “Index Merge Optimization”), or attempts to find the most restrictive index by deciding which index excludes more rows and using that index to fetch the rows.

If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to look up rows. For example, if you have a three-column index on `(col1, col2, col3)`, you have indexed search capabilities on `(col1)`, `(col1, col2)`, and `(col1, col2, col3)`.

MySQL cannot use the index to perform lookups if the columns do not form a leftmost prefix of the index. Suppose that you have the `SELECT` statements shown here:

```
SELECT * FROM tbl_name WHERE col1=val1;
SELECT * FROM tbl_name WHERE col1=val1 AND col2=val2;

SELECT * FROM tbl_name WHERE col2=val2;
SELECT * FROM tbl_name WHERE col2=val2 AND col3=val3;
```

If an index exists on `(col1, col2, col3)`, only the first two queries use the index. The third and fourth queries do involve indexed columns, but do not use an index to perform lookups because `(col2)` and `(col2, col3)` are not leftmost prefixes of `(col1, col2, col3)`.

8.3.7 Verifying Index Usage

Always check whether all your queries really use the indexes that you have created in the tables. Use the `EXPLAIN` statement, as described in Section 8.8.1, “Optimizing Queries with EXPLAIN”.

8.3.8 InnoDB and MyISAM Index Statistics Collection

Storage engines collect statistics about tables for use by the optimizer. Table statistics are based on value groups, where a value group is a set of rows with the same key prefix value. For optimizer purposes, an important statistic is the average value group size.

MySQL uses the average value group size in the following ways:

- To estimate how many rows must be read for each `ref` access
- To estimate how many rows a partial join produces, that is, the number of rows produced by an operation of the form

```
(...) JOIN tbl_name ON tbl_name.key = expr
```

As the average value group size for an index increases, the index is less useful for those two purposes because the average number of rows per lookup increases: For the index to be good for optimization purposes, it is best that each index value target a small number of rows in the table. When a given index value yields a large number of rows, the index is less useful and MySQL is less likely to use it.

The average value group size is related to table cardinality, which is the number of value groups. The `SHOW INDEX` statement displays a cardinality value based on *`N/S`*, where *`N`* is the number of rows in the table and *`S`* is the average value group size. That ratio yields an approximate number of value groups in the table.

For a join based on the `<=>` comparison operator, `NULL` is not treated differently from any other value: `NULL <=> NULL`, just as *`N <=> N`* for any other *`N`*.

However, for a join based on the `=` operator, `NULL` is different from non-`NULL` values: *`expr1 = expr2`* is not true when *`expr1`* or *`expr2`* (or both) are `NULL`. This affects `ref` accesses for comparisons of the form *`tbl_name.key = expr`*: MySQL does not access the table if the current value of *`expr`* is `NULL`, because the comparison cannot be true.

For `=` comparisons, it does not matter how many `NULL` values are in the table. For optimization purposes, the relevant value is the average size of the non-`NULL` value groups. However, MySQL does not currently enable that average size to be collected or used.

For `InnoDB` and `MyISAM` tables, you have some control over collection of table statistics by means of the `innodb_stats_method` and `myisam_stats_method` system variables, respectively. These variables have three possible values, which differ as follows:

- When the variable is set to `nulls_equal`, all `NULL` values are treated as identical (that is, they all form a single value group).

  If the `NULL` value group size is much higher than the average non-`NULL` value group size, this method skews the average value group size upward. This makes index appear to the optimizer to be less useful than it really is for joins that look for non-`NULL` values. Consequently, the `nulls_equal` method may cause the optimizer not to use the index for `ref` accesses when it should.

- When the variable is set to `nulls_unequal`, `NULL` values are not considered the same. Instead, each `NULL` value forms a separate value group of size 1.

  If you have many `NULL` values, this method skews the average value group size downward. If the average non-`NULL` value group size is large, counting `NULL` values each as a group of size 1 causes the optimizer to overestimate the value of the index for joins that look for non-`NULL` values. Consequently, the `nulls_unequal` method may cause the optimizer to use this index for `ref` lookups when other methods may be better.

- When the variable is set to `nulls_ignored`, `NULL` values are ignored.

If you tend to use many joins that use `<=>` rather than `=`, `NULL` values are not special in comparisons and one `NULL` is equal to another. In this case, `nulls_equal` is the appropriate statistics method.