GTID-based replication, check for and clear all ignored server ID lists on the servers involved. The `SHOW REPLICA STATUS` statement displays the list of ignored IDs, if there is one. If you do receive the deprecation warning, you can still clear a list after `gtid_mode=ON` is set by issuing a `CHANGE MASTER TO` statement containing the `IGNORE_SERVER_IDS` option with an empty list.

`MASTER_AUTO_POSITION = {0|1}`

Makes the replica attempt to connect to the source using the auto-positioning feature of GTID-based replication, rather than a binary log file based position. This option is used to start a replica using GTID-based replication. The default is 0, meaning that GTID auto-positioning and GTID-based replication are not used. This option can be used with `CHANGE MASTER TO` only if both the replication SQL (applier) thread and replication I/O (receiver) thread are stopped.

Both the replica and the source must have GTIDs enabled (`GTID_MODE=ON`, `ON_PERMISSIVE`, or `OFF_PERMISSIVE` on the replica, and `GTID_MODE=ON` on the source). `MASTER_LOG_FILE`, `MASTER_LOG_POS`, `RELAY_LOG_FILE`, and `RELAY_LOG_POS` cannot be specified together with `MASTER_AUTO_POSITION = 1`. If multi-source replication is enabled on the replica, you need to set the `MASTER_AUTO_POSITION = 1` option for each applicable replication channel.

With `MASTER_AUTO_POSITION = 1` set, in the initial connection handshake, the replica sends a GTID set containing the transactions that it has already received, committed, or both. The source responds by sending all transactions recorded in its binary log whose GTID is not included in the GTID set sent by the replica. This exchange ensures that the source only sends the transactions with a GTID that the replica has not already recorded or committed. If the replica receives transactions from more than one source, as in the case of a diamond topology, the auto-skip function ensures that the transactions are not applied twice. For details of how the GTID set sent by the replica is computed, see Section 17.1.3.3, “GTID Auto-Positioning”.

If any of the transactions that should be sent by the source have been purged from the source's binary log, or added to the set of GTIDs in the `gtid_purged` system variable by another method, the source sends the error `ER_MASTER_HAS_PURGED_REQUIRED_GTIDS` to the replica, and replication does not start. The GTIDs of the missing purged transactions are identified and listed in the source's error log in the warning message `ER_FOUND_MISSING_GTIDS`. Also, if during the exchange of transactions it is found that the replica has recorded or committed transactions with the source's UUID in the GTID, but the source itself has not committed them, the source sends the error `ER_SLAVE_HAS_MORE_GTIDS_THAN_MASTER` to the replica and replication does not start. For information on how to handle these situations, see Section 17.1.3.3, “GTID Auto-Positioning”.

You can see whether replication is running with GTID auto-positioning enabled by checking the Performance Schema

`replication_connection_status` table or the output of `SHOW REPLICA STATUS`. Disabling the `MASTER_AUTO_POSITION` option again makes the replica revert to file-based replication.

`MASTER_BIND = 'interface_name'`

Determines which of the replica's network interfaces is chosen for connecting to the source, for use on replicas that have multiple network interfaces. Specify the IP address of the network interface. The maximum length of the string value is 255 characters.

The IP address configured with this option, if any, can be seen in the `Master_Bind` column of the output from `SHOW REPLICA STATUS`. In the source metadata repository table `mysql.slave_master_info`, the value can be seen as the `Master_bind` column. The ability to bind a replica to a specific network interface is also supported by NDB Cluster.

`MASTER_COMPRESSION_ALGORITHMS = 'algorithm[,algorithm][,algorithm]'`

Specifies one, two, or three of the permitted compression algorithms for connections to the replication source server, separated by commas. The maximum length of the string value is 99 characters. The default value is `uncompressed`.

The available algorithms are `zlib`, `zstd`, and `uncompressed`, the same as for the `protocol_compression_algorithms` system variable. The algorithms can be specified in any order, but it is not an order of preference - the algorithm negotiation process attempts to use `zlib`, then `zstd`, then `uncompressed`, if they are specified. `MASTER_COMPRESSION_ALGORITHMS` is available as of MySQL 8.0.18.

The value of `MASTER_COMPRESSION_ALGORITHMS` applies only if the `replica_compressed_protocol` or `slave_compressed_protocol` system variable is disabled. If `replica_compressed_protocol` or `slave_compressed_protocol` is enabled, it takes precedence over `MASTER_COMPRESSION_ALGORITHMS` and connections to the source use `zlib` compression if both source and replica support that algorithm. For more information, see Section 4.2.8, “Connection Compression Control”.

Binary log transaction compression (available as of MySQL 8.0.20), which is activated by the `binlog_transaction_compression` system variable, can also be used to save bandwidth. If you do this in combination with connection compression, connection compression has less opportunity to act on the data, but can still compress headers and those events and transaction payloads that are uncompressed. For more information on binary log transaction compression, see Section 5.4.4.5, “Binary Log Transaction Compression”.

`MASTER_CONNECT_RETRY = interval`

Specifies the interval in seconds between the reconnection attempts that the replica makes after the connection to the source times out. The default interval is 60.

The attempts are limited by the `MASTER_RETRY_COUNT` option. If both the default settings are used, the replica waits 60 seconds between reconnection attempts (`MASTER_CONNECT_RETRY=60`), and keeps attempting to reconnect at this rate for 60 days (`MASTER_RETRY_COUNT=86400`). These values are recorded in the source metadata repository and shown in the

`replication_connection_configuration` Performance Schema table.

`MASTER_DELAY = interval`

Specifies how many seconds behind the source the replica must lag. An event received from the source is not executed until at least *`interval`* seconds later than its execution on the source. *`interval`* must be a nonnegative integer in the range from 0 to $2^{31}-1$. The default is 0. For more information, see Section 17.4.11, “Delayed Replication”.

A `CHANGE MASTER TO` statement employing the `MASTER_DELAY` option can be executed on a running replica when the replication SQL thread is stopped.

`MASTER_HEARTBEAT_PERIOD = interval`

Controls the heartbeat interval, which stops the connection timeout occurring in the absence of data if the connection is still good. A heartbeat signal is sent to the replica after that number of seconds, and the waiting period is reset whenever the source's binary log is updated with an event. Heartbeats are therefore sent by the source only if there are no unsent events in the binary log file for a period longer than this.

The heartbeat interval *`interval`* is a decimal value having the range 0 to 4294967 seconds and a resolution in milliseconds; the smallest nonzero value is 0.001. Setting *`interval`* to 0 disables heartbeats altogether. The heartbeat interval defaults to half the value of the `replica_net_timeout` or `slave_net_timeout` system variable. It is recorded in the source metadata repository and shown in the `replication_connection_configuration` Performance Schema table.

The system variable `replica_net_timeout` (from MySQL 8.0.26) or `slave_net_timeout` (before MySQL 8.0.26) specifies the number of seconds that the replica waits for either more data or a heartbeat signal from the source, before the replica considers the connection broken, aborts the read, and tries to reconnect. The default value is 60 seconds (one minute). Note that a change to the value or default setting of `replica_net_timeout` or `slave_net_timeout` does not automatically change the heartbeat interval, whether that has been set explicitly or is using a previously calculated default. A warning is issued if you set the global value of `replica_net_timeout` or `slave_net_timeout` to a value less than that of the current heartbeat interval. If `replica_net_timeout` or `slave_net_timeout` is changed, you must also issue `CHANGE MASTER TO` to adjust the heartbeat interval to an appropriate value so that the heartbeat signal occurs before the connection timeout. If you do not do this, the heartbeat signal has no effect, and if no data is received from the source, the replica can make repeated reconnection attempts, creating zombie dump threads.

`MASTER_HOST = 'host_name'`

The host name or IP address of the replication source server. The replica uses this to connect to the source. The maximum length of the string value is 255 characters. Before MySQL 8.0.17 it was 60 characters.

If you specify `MASTER_HOST` or `MASTER_PORT`, the replica assumes that the source server is different from before (even if the option value is the same as its current value.) In this case, the old values

for the source's binary log file name and position are considered no longer applicable, so if you do not specify `MASTER_LOG_FILE` and `MASTER_LOG_POS` in the statement, `MASTER_LOG_FILE=''` and `MASTER_LOG_POS=4` are silently appended to it.

Setting `MASTER_HOST=''` (that is, setting its value explicitly to an empty string) is *not* the same as not setting `MASTER_HOST` at all. Trying to set `MASTER_HOST` to an empty string fails with an error.

`MASTER_LOG_FILE = 'source_log_name', MASTER_LOG_POS = source_log_pos`

The binary log file name, and the location in that file, at which the replication I/O (receiver) thread begins reading from the source's binary log the next time the thread starts. Specify these options if you are using binary log file position based replication.

`MASTER_LOG_FILE` must include the numeric suffix of a specific binary log file that is available on the source server, for example, `MASTER_LOG_FILE='binlog.000145'`. The maximum length of the string value is 511 characters.

`MASTER_LOG_POS` is the numeric position for the replica to start reading in that file. `MASTER_LOG_POS=4` represents the start of the events in a binary log file.

If you specify either of `MASTER_LOG_FILE` or `MASTER_LOG_POS`, you cannot specify `MASTER_AUTO_POSITION = 1`, which is for GTID-based replication.

If neither of `MASTER_LOG_FILE` or `MASTER_LOG_POS` is specified, the replica uses the last coordinates of the *replication SQL (applier) thread* before `CHANGE MASTER TO` was issued. This ensures that there is no discontinuity in replication, even if the replication SQL (applier) thread was late compared to the replication I/O (receiver) thread.

`MASTER_PASSWORD = 'password'`

The password for the replication user account to use for connecting to the replication source server. The maximum length of the string value is 32 characters. If you specify `MASTER_PASSWORD`, `MASTER_USER` is also required.

The password used for a replication user account in a `CHANGE MASTER TO` statement is limited to 32 characters in length. Trying to use a password of more than 32 characters causes `CHANGE MASTER TO` to fail.

The password is masked in MySQL Server's logs, Performance Schema tables, and `SHOW PROCESSLIST` statements.

`MASTER_PORT = port_num`

The TCP/IP port number that the replica uses to connect to the replication source server.

Note

Replication cannot use Unix socket files. You must be able to connect to the replication source server using TCP/IP.

If you specify `MASTER_HOST` or `MASTER_PORT`, the replica assumes that the source server is different from before (even if the option value is the same as its current value). In this case, the old values for the source's binary log file name and position are considered no

longer applicable, so if you do not specify `MASTER_LOG_FILE` and `MASTER_LOG_POS` in the statement, `MASTER_LOG_FILE=''` and `MASTER_LOG_POS=4` are silently appended to it.

`MASTER_PUBLIC_KEY_PATH = 'key_file_name'`

Enables RSA key pair-based password exchange by providing the path name to a file containing a replica-side copy of the public key required by the source. The file must be in PEM format. The maximum length of the string value is 511 characters.

This option applies to replicas that authenticate with the `sha256_password` or `caching_sha2_password` authentication plugin. (For `sha256_password`, `MASTER_PUBLIC_KEY_PATH` can be used only if MySQL was built using OpenSSL.) If you are using a replication user account that authenticates with the `caching_sha2_password` plugin (which is the default from MySQL 8.0), and you are not using a secure connection, you must specify either this option or the `GET_MASTER_PUBLIC_KEY=1` option to provide the RSA public key to the replica.

`MASTER_RETRY_COUNT = count`

Sets the maximum number of reconnection attempts that the replica makes after the connection to the source times out, as determined by the `replica_net_timeout` or `slave_net_timeout` system variable. If the replica does need to reconnect, the first retry occurs immediately after the timeout. The default is 86400 attempts.

The interval between the attempts is specified by the `MASTER_CONNECT_RETRY` option. If both the default settings are used, the replica waits 60 seconds between reconnection attempts (`MASTER_CONNECT_RETRY=60`), and keeps attempting to reconnect at this rate for 60 days (`MASTER_RETRY_COUNT=86400`). A setting of 0 for `MASTER_RETRY_COUNT` means that there is no limit on the number of reconnection attempts, so the replica keeps trying to reconnect indefinitely.

The values for `MASTER_CONNECT_RETRY` and `MASTER_RETRY_COUNT` are recorded in the source metadata repository and shown in the `replication_connection_configuration` Performance Schema table. `MASTER_RETRY_COUNT` supersedes the `--master-retry-count` server startup option.

`MASTER_SSL = {0|1}`

Specify whether the replica encrypts the replication connection. The default is 0, meaning that the replica does not encrypt the replication connection. If you set `MASTER_SSL=1`, you can configure the encryption using the `MASTER_SSL_xxx` and `MASTER_TLS_xxx` options.

Setting `MASTER_SSL=1` for a replication connection and then setting no further `MASTER_SSL_xxx` options corresponds to setting `--ssl-mode=REQUIRED` for the client, as described in Command Options for Encrypted Connections. With `MASTER_SSL=1`, the connection attempt only succeeds if an encrypted connection can be established. A replication connection does not fall back to an unencrypted connection, so there is no setting corresponding to the

`--ssl-mode=PREFERRED` setting for replication. If `MASTER_SSL=0` is set, this corresponds to `--ssl-mode=DISABLED`.

Important

To help prevent sophisticated man-in-the-middle attacks, it is important for the replica to verify the server's identity. You can specify additional `MASTER_SSL_xxx` options to correspond to the settings `--ssl-mode=VERIFY_CA` and `--ssl-mode=VERIFY_IDENTITY`, which are a better choice than the default setting to help prevent this type of attack. With these settings, the replica checks that the server's certificate is valid, and checks that the host name the replica is using matches the identity in the server's certificate. To implement one of these levels of verification, you must first ensure that the CA certificate for the server is reliably available to the replica, otherwise availability issues will result. For this reason, they are not the default setting.

`MASTER_SSL_xxx`, `MASTER_TLS_xxx`

Specify how the replica uses encryption and ciphers to secure the replication connection. These options can be changed even on replicas that are compiled without SSL support. They are saved to the source metadata repository, but are ignored if the replica does not have SSL support enabled. The maximum length of the value for the string-valued `MASTER_SSL_xxx` and `MASTER_TLS_xxx` options is 511 characters, with the exception of `MASTER_TLS_CIPHERSUITES`, for which it is 4000 characters.

The `MASTER_SSL_xxx` and `MASTER_TLS_xxx` options perform the same functions as the `--ssl-xxx` and `--tls-xxx` client options described in Command Options for Encrypted Connections. The correspondence between the two sets of options, and the use of the `MASTER_SSL_xxx` and `MASTER_TLS_xxx` options to set up a secure connection, is explained in Section 17.3.1, "Setting Up Replication to Use Encrypted Connections".

`MASTER_USER = 'user_name'`

The user name for the replication user account to use for connecting to the replication source server. The maximum length of the string value is 96 characters.

For Group Replication, this account must exist on every member of the replication group. It is used for distributed recovery if the XCom communication stack is in use for the group, and also used for group communication connections if the MySQL communication stack is in use for the group. With the MySQL communication stack, the account must have the `GROUP_REPLICATION_STREAM` permission.

It is possible to set an empty user name by specifying `MASTER_USER=''`, but the replication channel cannot be started with an empty user name. In releases before MySQL 8.0.21, only set an empty `MASTER_USER` user name if you need to clear previously used credentials from the replication metadata repositories for security purposes. Do not use the channel

afterwards, due to a bug in these releases that can substitute a default user name if an empty user name is read from the repositories (for example, during an automatic restart of a Group Replication channel). From MySQL 8.0.21, it is valid to set an empty `MASTER_USER` user name and use the channel afterwards if you always provide user credentials using the `START REPLICA` statement or `START GROUP_REPLICATION` statement that starts the replication channel. This approach means that the replication channel always needs operator intervention to restart, but the user credentials are not recorded in the replication metadata repositories.

Important

To connect to the source using a replication user account that authenticates with the `caching_sha2_password` plugin, you must either set up a secure connection as described in Section 17.3.1, “Setting Up Replication to Use Encrypted Connections”, or enable the unencrypted connection to support password exchange using an RSA key pair. The `caching_sha2_password` authentication plugin is the default for new users created from MySQL 8.0 (for details, see Section 6.4.1.2, “Caching SHA-2 Pluggable Authentication”). If the user account that you create or use for replication uses this authentication plugin, and you are not using a secure connection, you must enable RSA key pair-based password exchange for a successful connection. You can do this using either the `MASTER_PUBLIC_KEY_PATH` option or the `GET_MASTER_PUBLIC_KEY=1` option for this statement.

`MASTER_ZSTD_COMPRESSION_LEVEL = level`

The compression level to use for connections to the replication source server that use the `zstd` compression algorithm. The permitted levels are from 1 to 22, with larger values indicating increasing levels of compression. The default level is 3. `MASTER_ZSTD_COMPRESSION_LEVEL` is available as of MySQL 8.0.18.

The compression level setting has no effect on connections that do not use `zstd` compression. For more information, see Section 4.2.8, “Connection Compression Control”.

`NETWORK_NAMESPACE = 'namespace'`

The network namespace to use for TCP/IP connections to the replication source server or, if the MySQL communication stack is in use, for Group Replication’s group communication connections. The maximum length of the string value is 64 characters. If this option is omitted, connections from the replica use the default (global) namespace. On platforms that do not implement network namespace support, failure occurs when the replica attempts to connect to the source. For information about network namespaces, see Section 5.1.14, “Network Namespace Support”. `NETWORK_NAMESPACE` is available as of MySQL 8.0.22.

`PRIVILEGE_CHECKS_USER = {NULL | 'account'}`

Names a user account that supplies a security context for the specified channel. `NULL`, which is the default, means no security context is used. `PRIVILEGE_CHECKS_USER` is available as of MySQL 8.0.18.

The user name and host name for the user account must follow the syntax described in Section 6.2.4, “Specifying Account Names”, and the user must not be an anonymous user (with a blank user name) or the `CURRENT_USER`. The account must have the `REPLICATION_APPLIER` privilege, plus the required privileges to execute the transactions replicated on the channel. For details of the privileges required by the account, see Section 17.3.3, “Replication Privilege Checks”. When you restart the replication channel, the privilege checks are applied from that point on. If you do not specify a channel and no other channels exist, the statement is applied to the default channel.

The use of row-based binary logging is strongly recommended when `PRIVILEGE_CHECKS_USER` is set, and you can set `REQUIRE_ROW_FORMAT` to enforce this. For example, to start privilege checks on the channel `channel_1` on a running replica, issue the following statements:

```
mysql> STOP REPLICA FOR CHANNEL 'channel_1';
mysql> CHANGE MASTER TO
         PRIVILEGE_CHECKS_USER = 'priv_repl'@'%.example.com',
         REQUIRE_ROW_FORMAT = 1,
         FOR CHANNEL 'channel_1';
mysql> START REPLICA FOR CHANNEL 'channel_1';
```

For releases from MySQL 8.0.22, use `START REPLICA` and `STOP REPLICA`, and for releases before MySQL 8.0.22, use `START SLAVE` and `STOP SLAVE`. The statements work in the same way, only the terminology has changed.

`RELAY_LOG_FILE = 'relay_log_file'`, `RELAY_LOG_POS = 'relay_log_pos'`

The relay log file name, and the location in that file, at which the replication SQL thread begins reading from the replica's relay log the next time the thread starts. `RELAY_LOG_FILE` can use either an absolute or relative path, and uses the same base name as `MASTER_LOG_FILE`. The maximum length of the string value is 511 characters.

A `CHANGE MASTER TO` statement using `RELAY_LOG_FILE`, `RELAY_LOG_POS`, or both options can be executed on a running replica when the replication SQL thread is stopped. Relay logs are preserved if at least one of the replication SQL (applier) thread and the replication I/O (receiver) thread is running. If both threads are stopped, all relay log files are deleted unless at least one of `RELAY_LOG_FILE` or `RELAY_LOG_POS` is specified. For the Group Replication applier channel (`group_replication_applier`), which only has an applier thread and no receiver thread, this is the case if the applier thread is stopped, but with that channel you cannot use the `RELAY_LOG_FILE` and `RELAY_LOG_POS` options.

`REQUIRE_ROW_FORMAT = {0|1}`

Permits only row-based replication events to be processed by the replication channel. This option prevents the replication applier from taking actions such as creating temporary tables and executing `LOAD DATA INFILE` requests, which increases the security of the channel. The `REQUIRE_ROW_FORMAT` option is disabled by default for asynchronous replication channels, but it is enabled by default

for Group Replication channels, and it cannot be disabled for them. For more information, see Section 17.3.3, “Replication Privilege Checks”. `REQUIRE_ROW_FORMAT` is available as of MySQL 8.0.19.

`REQUIRE_TABLE_PRIMARY_KEY_CHECK = {STREAM | ON | OFF}`

Enables a replica to select its own policy for primary key checks. The default is `STREAM`. `REQUIRE_TABLE_PRIMARY_KEY_CHECK` is available as of MySQL 8.0.20.

When the option is set to `ON` for a replication channel, the replica always uses the value `ON` for the `sql_require_primary_key` system variable in replication operations, requiring a primary key. When the option is set to `OFF`, the replica always uses the value `OFF` for the `sql_require_primary_key` system variable in replication operations, so that a primary key is never required, even if the source required one. When the `REQUIRE_TABLE_PRIMARY_KEY_CHECK` option is set to `STREAM`, which is the default, the replica uses whatever value is replicated from the source for each transaction.

For multisource replication, setting `REQUIRE_TABLE_PRIMARY_KEY_CHECK` to `ON` or `OFF` enables a replica to normalize behavior across the replication channels for different sources, and keep a consistent setting for the `sql_require_primary_key` system variable. Using `ON` safeguards against the accidental loss of primary keys when multiple sources update the same set of tables. Using `OFF` allows sources that can manipulate primary keys to work alongside sources that cannot.

When `PRIVILEGE_CHECKS_USER` is set, setting `REQUIRE_TABLE_PRIMARY_KEY_CHECK` to `ON` or `OFF` means that the user account does not need session administration level privileges to set restricted session variables, which are required to change the value of `sql_require_primary_key` to match the source's setting for each transaction. For more information, see Section 17.3.3, “Replication Privilege Checks”.

`SOURCE_CONNECTION_AUTO_FAILOVER = {0|1}`

Activates the asynchronous connection failover mechanism for a replication channel if one or more alternative replication source servers are available (so when there are multiple MySQL servers or groups of servers that share the replicated data). `SOURCE_CONNECTION_AUTO_FAILOVER` is available as of MySQL 8.0.22. The default is 0, meaning that the mechanism is not activated. For full information and instructions to set up this feature, see Section 17.4.9.2, “Asynchronous Connection Failover for Replicas”.

The asynchronous connection failover mechanism takes over after the reconnection attempts controlled by `MASTER_CONNECT_RETRY` and `MASTER_RETRY_COUNT` are exhausted. It reconnects the replica to an alternative source chosen from a specified source list, which you manage using the `asynchronous_connection_failover_add_source` and `asynchronous_connection_failover_delete_source` functions. To add and remove managed groups of servers, use the `asynchronous_connection_failover_add_managed` and `asynchronous_connection_failover_delete_managed` functions instead. For more information, see Section 17.4.9,

"Switching Sources and Replicas with Asynchronous Connection Failover".

Important

1. You can only set `SOURCE_CONNECTION_AUTO_FAILOVER = 1` when GTID auto-positioning is in use (`MASTER_AUTO_POSITION = 1`).

2. When you set `SOURCE_CONNECTION_AUTO_FAILOVER = 1`, set `MASTER_RETRY_COUNT` and `MASTER_CONNECT_RETRY` to minimal numbers that just allow a few retry attempts with the same source in a short time, in case the connection failure is caused by a transient network outage. Otherwise the asynchronous connection failover mechanism cannot be activated promptly. Suitable values are `MASTER_RETRY_COUNT=3` and `MASTER_CONNECT_RETRY=10`, which make the replica retry the connection 3 times with 10-second intervals between.

3. When you set `SOURCE_CONNECTION_AUTO_FAILOVER = 1`, the replication metadata repositories must contain the credentials for a replication user account that can be used to connect to all the servers on the source list for the replication channel. These credentials can be set using the `CHANGE REPLICATION SOURCE TO` statement with the `MASTER_USER` and `MASTER_PASSWORD` options. For more information, see Section 17.4.9, "Switching Sources and Replicas with Asynchronous Connection Failover".

4. From MySQL 8.0.27, when you set `SOURCE_CONNECTION_AUTO_FAILOVER = 1`, asynchronous connection failover for replicas is automatically activated if this replication channel is on a Group Replication primary in a group in single-primary mode. With this function active, if the primary that is replicating goes offline or into an error state, the new primary starts replication on the same channel when it is elected. If you want to use the function, this replication channel must also be set up on all the secondary servers in the replication group, and on any new joining members. (If the servers are provisioned using MySQL's clone functionality, this all happens

automatically.) If you do not want to use the function, disable it by using the `group_replication_disable_member_action` function to disable the Group Replication member action `mysql_start_failover_channels_if_primary` which is enabled by default. For more information, see Section 17.4.9.2, “Asynchronous Connection Failover for Replicas”.

Examples

`CHANGE MASTER TO` is useful for setting up a replica when you have the snapshot of the source and have recorded the source's binary log coordinates corresponding to the time of the snapshot. After loading the snapshot into the replica to synchronize it with the source, you can run `CHANGE MASTER TO MASTER_LOG_FILE='log_name', MASTER_LOG_POS=log_pos` on the replica to specify the coordinates at which the replica should begin reading the source's binary log. The following example changes the source server the replica uses and establishes the source's binary log coordinates from which the replica begins reading:

```
CHANGE MASTER TO
  MASTER_HOST='source2.example.com',
  MASTER_USER='replication',
  MASTER_PASSWORD='password',
  MASTER_PORT=3306,
  MASTER_LOG_FILE='source2-bin.001',
  MASTER_LOG_POS=4,
  MASTER_CONNECT_RETRY=10;
```

For the procedure to switch an existing replica to a new source during failover, see Section 17.4.8, “Switching Sources During Failover”.

When GTIDs are in use on the source and the replica, specify GTID auto-positioning instead of giving the binary log file position, as in the following example. For full instructions to configure and start GTID-based replication on new or stopped servers, online servers, or additional replicas, see Section 17.1.3, “Replication with Global Transaction Identifiers”.

```
CHANGE MASTER TO
  MASTER_HOST='source3.example.com',
  MASTER_USER='replication',
  MASTER_PASSWORD='password',
  MASTER_PORT=3306,
  MASTER_AUTO_POSITION = 1,
  FOR CHANNEL "source_3";
```

In this example, multi-source replication is in use, and the `CHANGE MASTER TO` statement is applied to the replication channel `"source_3"` that connects the replica to the specified host. For guidance on setting up multi-source replication, see Section 17.1.5, “MySQL Multi-Source Replication”.

The next example shows how to make the replica apply transactions from relay log files that you want to repeat. To do this, the source need not be reachable. You can use `CHANGE MASTER TO` to locate the relay log position where you want the replica to start reapplying transactions, and then start the SQL thread:

```
CHANGE MASTER TO
  RELAY_LOG_FILE='replica-relay-bin.006',
  RELAY_LOG_POS=4025;
START SLAVE SQL_THREAD;
```

`CHANGE MASTER TO` can also be used to skip over transactions in the binary log that are causing replication to stop. The appropriate method to do this depends on whether GTIDs are in use or not. For

instructions to skip transactions using `CHANGE MASTER TO` or another method, see Section 17.1.7.3, “Skipping Transactions”.

13.4.2.2 CHANGE REPLICATION FILTER Statement

```
CHANGE REPLICATION FILTER filter[, filter]
 [, ...] [FOR CHANNEL channel]

filter: {
    REPLICATE_DO_DB = (db_list)
  | REPLICATE_IGNORE_DB = (db_list)
  | REPLICATE_DO_TABLE = (tbl_list)
  | REPLICATE_IGNORE_TABLE = (tbl_list)
  | REPLICATE_WILD_DO_TABLE = (wild_tbl_list)
  | REPLICATE_WILD_IGNORE_TABLE = (wild_tbl_list)
  | REPLICATE_REWRITE_DB = (db_pair_list)
}

db_list:
    db_name[, db_name][, ...]

tbl_list:
    db_name.table_name[, db_name.table_name][, ...]
wild_tbl_list:
    'db_pattern.table_pattern'[, 'db_pattern.table_pattern'][, ...]

db_pair_list:
    (db_pair)[, (db_pair)][, ...]

db_pair:
    from_db, to_db
```

`CHANGE REPLICATION FILTER` sets one or more replication filtering rules on the replica in the same way as starting the replica `mysqld` with replication filtering options such as `--replicate-do-db` or `--replicate-wild-ignore-table`. Filters set using this statement differ from those set using the server options in two key respects:

1. The statement does not require restarting the server to take effect, only that the replication SQL thread be stopped using `STOP REPLICA SQL_THREAD` first (and restarted with `START REPLICA SQL_THREAD` afterwards).

2. The effects of the statement are not persistent; any filters set using `CHANGE REPLICATION FILTER` are lost following a restart of the replica `mysqld`.

`CHANGE REPLICATION FILTER` requires the `REPLICATION_SLAVE_ADMIN` privilege (or the deprecated `SUPER` privilege).

Use the `FOR CHANNEL channel` clause to make a replication filter specific to a replication channel, for example on a multi-source replica. Filters applied without a specific `FOR CHANNEL` clause are considered global filters, meaning that they are applied to all replication channels.

Note

Global replication filters cannot be set on a MySQL server instance that is configured for Group Replication, because filtering transactions on some servers would make the group unable to reach agreement on a consistent state. Channel specific replication filters can be set on replication channels that are not directly involved with Group Replication, such as where a group member also acts as a replica to a source that is outside the group. They cannot be set on the `group_replication_applier` or `group_replication_recovery` channels.

The following list shows the `CHANGE REPLICATION FILTER` options and how they relate to `--replicate-*` server options:

- `REPLICATE_DO_DB`: Include updates based on database name. Equivalent to `--replicate-do-db`.
- `REPLICATE_IGNORE_DB`: Exclude updates based on database name. Equivalent to `--replicate-ignore-db`.
- `REPLICATE_DO_TABLE`: Include updates based on table name. Equivalent to `--replicate-do-table`.
- `REPLICATE_IGNORE_TABLE`: Exclude updates based on table name. Equivalent to `--replicate-ignore-table`.
- `REPLICATE_WILD_DO_TABLE`: Include updates based on wildcard pattern matching table name. Equivalent to `--replicate-wild-do-table`.
- `REPLICATE_WILD_IGNORE_TABLE`: Exclude updates based on wildcard pattern matching table name. Equivalent to `--replicate-wild-ignore-table`.
- `REPLICATE_REWRITE_DB`: Perform updates on replica after substituting new name on replica for specified database on source. Equivalent to `--replicate-rewrite-db`.

The precise effects of `REPLICATE_DO_DB` and `REPLICATE_IGNORE_DB` filters are dependent on whether statement-based or row-based replication is in effect. See Section 17.2.5, “How Servers Evaluate Replication Filtering Rules”, for more information.

Multiple replication filtering rules can be created in a single `CHANGE REPLICATION FILTER` statement by separating the rules with commas, as shown here:

```
CHANGE REPLICATION FILTER
    REPLICATE_DO_DB = (d1), REPLICATE_IGNORE_DB = (d2);
```

Issuing the statement just shown is equivalent to starting the replica `mysqld` with the options `--replicate-do-db=d1 --replicate-ignore-db=d2`.

On a multi-source replica, which uses multiple replication channels to process transaction from different sources, use the `FOR CHANNEL` *`channel`* clause to set a replication filter on a replication channel:

```
CHANGE REPLICATION FILTER REPLICATE_DO_DB = (d1) FOR CHANNEL channel_1;
```

This enables you to create a channel specific replication filter to filter out selected data from a source. When a `FOR CHANNEL` clause is provided, the replication filter statement acts on that replication channel, removing any existing replication filter which has the same filter type as the specified replication filters, and replacing them with the specified filter. Filter types not explicitly listed in the statement are not modified. If issued against a replication channel which is not configured, the statement fails with an `ER_SLAVE_CONFIGURATION` error. If issued against Group Replication channels, the statement fails with an `ER_SLAVE_CHANNEL_OPERATION_NOT_ALLOWED` error.

On a replica with multiple replication channels configured, issuing `CHANGE REPLICATION FILTER` with no `FOR CHANNEL` clause configures the replication filter for every configured replication channel, and for the global replication filters. For every filter type, if the filter type is listed in the statement, then any existing filter rules of that type are replaced by the filter rules specified in the most recently issued statement, otherwise the old value of the filter type is retained. For more information see Section 17.2.5.4, “Replication Channel Based Filters”.

If the same filtering rule is specified multiple times, only the *last* such rule is actually used. For example, the two statements shown here have exactly the same effect, because the first `REPLICATE_DO_DB` rule in the first statement is ignored:

```
CHANGE REPLICATION FILTER
    REPLICATE_DO_DB = (db1, db2), REPLICATE_DO_DB = (db3, db4);

CHANGE REPLICATION FILTER
    REPLICATE_DO_DB = (db3, db4);
```

Caution

This behavior differs from that of the `--replicate-*` filter options where specifying the same option multiple times causes the creation of multiple filter rules.

Names of tables and database not containing any special characters need not be quoted. Values used with `REPLICATION_WILD_TABLE` and `REPLICATION_WILD_IGNORE_TABLE` are string expressions, possibly containing (special) wildcard characters, and so must be quoted. This is shown in the following example statements:

```
CHANGE REPLICATION FILTER
    REPLICATE_WILD_DO_TABLE = ('db1.old%');

CHANGE REPLICATION FILTER
    REPLICATE_WILD_IGNORE_TABLE = ('db1.new%', 'db2.new%');
```

Values used with `REPLICATE_REWRITE_DB` represent *pairs* of database names; each such value must be enclosed in parentheses. The following statement rewrites statements occurring on database `db1` on the source to database `db2` on the replica:

```
CHANGE REPLICATION FILTER REPLICATE_REWRITE_DB = ((db1, db2));
```

The statement just shown contains two sets of parentheses, one enclosing the pair of database names, and the other enclosing the entire list. This is perhaps more easily seen in the following example, which creates two `rewrite-db` rules, one rewriting database `dbA` to `dbB`, and one rewriting database `dbC` to `dbD`:

```
CHANGE REPLICATION FILTER
  REPLICATE_REWRITE_DB = ((dbA, dbB), (dbC, dbD));
```

The `CHANGE REPLICATION FILTER` statement replaces replication filtering rules only for the filter types and replication channels affected by the statement, and leaves other rules and channels unchanged. If you want to unset all filters of a given type, set the filter's value to an explicitly empty list, as shown in this example, which removes all existing `REPLICATE_DO_DB` and `REPLICATE_IGNORE_DB` rules:

```
CHANGE REPLICATION FILTER
    REPLICATE_DO_DB = (), REPLICATE_IGNORE_DB = ();
```

Setting a filter to empty in this way removes all existing rules, does not create any new ones, and does not restore any rules set at mysqld startup using `--replicate-*` options on the command line or in the configuration file.

The `RESET REPLICA ALL` statement removes channel specific replication filters that were set on channels deleted by the statement. When the deleted channel or channels are recreated, any global replication filters specified for the replica are copied to them, and no channel specific replication filters are applied.

For more information, see Section 17.2.5, “How Servers Evaluate Replication Filtering Rules”.

13.4.2.3 CHANGE REPLICATION SOURCE TO Statement

```
CHANGE REPLICATION SOURCE TO option [, option] ... [ channel_option ]

option: {
    SOURCE_BIND = 'interface_name'
  | SOURCE_HOST = 'host_name'
  | SOURCE_USER = 'user_name'
  | SOURCE_PASSWORD = 'password'
  | SOURCE_PORT = port_num
  | PRIVILEGE_CHECKS_USER = {NULL | 'account'}
  | REQUIRE_ROW_FORMAT = {0|1}
  | REQUIRE_TABLE_PRIMARY_KEY_CHECK = {STREAM | ON | OFF | GENERATE}
  | ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS = {OFF | LOCAL | uuid}
  | SOURCE_LOG_FILE = 'source_log_name'
```

```
  | SOURCE_LOG_POS = source_log_pos
  | SOURCE_AUTO_POSITION = {0|1}
  | RELAY_LOG_FILE = 'relay_log_name'
  | RELAY_LOG_POS = relay_log_pos
  | SOURCE_HEARTBEAT_PERIOD = interval
  | SOURCE_CONNECT_RETRY = interval
  | SOURCE_RETRY_COUNT = count
  | SOURCE_CONNECTION_AUTO_FAILOVER = {0|1}
  | SOURCE_DELAY = interval
  | SOURCE_COMPRESSION_ALGORITHMS = 'algorithm[,algorithm][,algorithm]'
  | SOURCE_ZSTD_COMPRESSION_LEVEL = level
  | SOURCE_SSL = {0|1}
  | SOURCE_SSL_CA = 'ca_file_name'
  | SOURCE_SSL_CAPATH = 'ca_directory_name'
  | SOURCE_SSL_CERT = 'cert_file_name'
  | SOURCE_SSL_CRL = 'crl_file_name'
  | SOURCE_SSL_CRLPATH = 'crl_directory_name'
  | SOURCE_SSL_KEY = 'key_file_name'
  | SOURCE_SSL_CIPHER = 'cipher_list'
  | SOURCE_SSL_VERIFY_SERVER_CERT = {0|1}
  | SOURCE_TLS_VERSION = 'protocol_list'
  | SOURCE_TLS_CIPHERSUITES = 'ciphersuite_list'
  | SOURCE_PUBLIC_KEY_PATH = 'key_file_name'
  | GET_SOURCE_PUBLIC_KEY = {0|1}
  | NETWORK_NAMESPACE = 'namespace'
  | IGNORE_SERVER_IDS = (server_id_list),
  | GTID_ONLY = {0|1}
}

channel_option:
    FOR CHANNEL channel

server_id_list:
    [server_id [, server_id] ... ]
```

`CHANGE REPLICATION SOURCE TO` changes the parameters that the replica server uses for connecting to the source and reading data from the source. It also updates the contents of the replication metadata repositories (see Section 17.2.4, “Relay Log and Replication Metadata Repositories”). In MySQL 8.0.23 and later, use `CHANGE REPLICATION SOURCE TO` in place of the deprecated `CHANGE MASTER TO` statement.

`CHANGE REPLICATION SOURCE TO` requires the `REPLICATION_SLAVE_ADMIN` privilege (or the deprecated `SUPER` privilege).

Options that you do not specify on a `CHANGE REPLICATION SOURCE TO` statement retain their value, except as indicated in the following discussion. In most cases, there is therefore no need to specify options that do not change.

Values used for `SOURCE_HOST` and other `CHANGE REPLICATION SOURCE TO` options are checked for linefeed (`\n` or `0x0A`) characters. The presence of such characters in these values causes the statement to fail with an error.

The optional `FOR CHANNEL channel` clause lets you name which replication channel the statement applies to. Providing a `FOR CHANNEL channel` clause applies the `CHANGE REPLICATION SOURCE TO` statement to a specific replication channel, and is used to add a new channel or modify an existing channel. For example, to add a new channel called `channel2`:

```
CHANGE REPLICATION SOURCE TO SOURCE_HOST=host1, SOURCE_PORT=3002 FOR CHANNEL 'channel2';
```

If no clause is named and no extra channels exist, a `CHANGE REPLICATION SOURCE TO` statement applies to the default channel, whose name is the empty string (""). When you have set up multiple replication channels, every `CHANGE REPLICATION SOURCE TO` statement must name a channel using the `FOR CHANNEL channel` clause. See Section 17.2.2, “Replication Channels” for more information.

For some of the options of the `CHANGE REPLICATION SOURCE TO` statement, you must issue a `STOP REPLICA` statement prior to issuing a `CHANGE REPLICATION SOURCE TO` statement (and

a START REPLICA statement afterwards). Sometimes, you only need to stop the replication SQL (applier) thread or the replication I/O (receiver) thread, not both:

- When the applier thread is stopped, you can execute CHANGE REPLICATION SOURCE TO using any combination that is otherwise allowed of RELAY_LOG_FILE, RELAY_LOG_POS, and SOURCE_DELAY options, even if the replication receiver thread is running. No other options may be used with this statement when the receiver thread is running.
- When the receiver thread is stopped, you can execute CHANGE REPLICATION SOURCE TO using any of the options for this statement (in any allowed combination) *except* RELAY_LOG_FILE, RELAY_LOG_POS, SOURCE_DELAY, or SOURCE_AUTO_POSITION = 1 even when the applier thread is running.
- Both the receiver thread and the applier thread must be stopped before issuing a CHANGE REPLICATION SOURCE TO statement that employs SOURCE_AUTO_POSITION = 1, GTID_ONLY = 1, or ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS.

You can check the current state of the replication applier thread and replication receiver thread using SHOW REPLICA STATUS. Note that the Group Replication applier channel (group_replication_applier) has no receiver thread, only an applier thread.

CHANGE REPLICATION SOURCE TO statements have a number of side-effects and interactions that you should be aware of beforehand:

- CHANGE REPLICATION SOURCE TO causes an implicit commit of an ongoing transaction. See Section 13.3.3, “Statements That Cause an Implicit Commit”.
- CHANGE REPLICATION SOURCE TO causes the previous values for SOURCE_HOST, SOURCE_PORT, SOURCE_LOG_FILE, and SOURCE_LOG_POS to be written to the error log, along with other information about the replica's state prior to execution.
- If you are using statement-based replication and temporary tables, it is possible for a CHANGE REPLICATION SOURCE TO statement following a STOP REPLICA statement to leave behind temporary tables on the replica. A warning (ER_WARN_OPEN_TEMP_TABLES_MUST_BE_ZERO) is issued whenever this occurs. You can avoid this in such cases by making sure that the value of the Replica_open_temp_tables or Slave_open_temp_tables system status variable is equal to 0 prior to executing such a CHANGE REPLICATION SOURCE TO statement.
- When using a multithreaded replica (replica_parallel_workers > 0), stopping the replica can cause gaps in the sequence of transactions that have been executed from the relay log, regardless of whether the replica was stopped intentionally or otherwise. When such gaps exist, issuing CHANGE REPLICATION SOURCE TO fails. The solution in this situation is to issue START REPLICA UNTIL SQL_AFTER_MTS_GAPS which ensures that the gaps are closed. From MySQL 8.0.26, the process of checking for gaps in the sequence of transactions is skipped entirely when GTID-based replication and GTID auto-positioning are in use, because gaps in transactions can be resolved using GTID auto-positioning. In that situation, CHANGE REPLICATION SOURCE TO can still be used.

The following options are available for CHANGE REPLICATION SOURCE TO statements:

ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS = {OFF | LOCAL | *uuid*}

Makes the replication channel assign a GTID to replicated transactions that do not have one, enabling replication from a source that does not use GTID-based replication, to a replica that does. For a multi-source replica, you can have a mix of channels that use ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS, and channels that do not. The default is OFF, meaning that the feature is not used.

LOCAL assigns a GTID including the replica's own UUID (the server_uuid setting). *uuid* assigns a GTID including the specified UUID, such as the server_uuid setting for the replication source server. Using a nonlocal UUID lets you

differentiate between transactions that originated on the replica and transactions that originated on the source, and for a multi-source replica, between transactions that originated on different sources. The UUID you choose only has significance for the replica's own use. If any of the transactions sent by the source do have a GTID already, that GTID is retained.

Channels specific to Group Replication cannot use `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS`, but an asynchronous replication channel for another source on a server instance that is a Group Replication group member can do so. In that case, do not specify the Group Replication group name as the UUID for creating the GTIDs.

To set `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` to `LOCAL` or *`uuid`*, the replica must have `gtid_mode=ON` set, and this cannot be changed afterwards. This option is for use with a source that has binary log file position based replication, so `SOURCE_AUTO_POSITION=1` cannot be set for the channel. Both the replication SQL thread and the replication I/O (receiver) thread must be stopped before setting this option.

Important

A replica set up with `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` on any channel cannot be promoted to replace the replication source server in the event that a failover is required, and a backup taken from the replica cannot be used to restore the replication source server. The same restriction applies to replacing or restoring other replicas that use `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` on any channel.

For further restrictions and information, see Section 17.1.3.6, “Replication From a Source Without GTIDs to a Replica With GTIDs”.

`GET_SOURCE_PUBLIC_KEY = {0|1}`

Enables RSA key pair-based password exchange by requesting the public key from the source. The option is disabled by default.

This option applies to replicas that authenticate with the `caching_sha2_password` authentication plugin. For connections by accounts that authenticate using this plugin, the source does not send the public key unless requested, so it must be requested or specified in the client. If `SOURCE_PUBLIC_KEY_PATH` is given and specifies a valid public key file, it takes precedence over `GET_SOURCE_PUBLIC_KEY`. If you are using a replication user account that authenticates with the `caching_sha2_password` plugin (which is the default from MySQL 8.0), and you are not using a secure connection, you must specify either this option or the `SOURCE_PUBLIC_KEY_PATH` option to provide the RSA public key to the replica.

`GTID_ONLY = {0|1}`

Stops the replication channel persisting file names and file positions in the replication metadata repositories. `GTID_ONLY` is available as of MySQL 8.0.27. The `GTID_ONLY` option is disabled by default for

asynchronous replication channels, but it is enabled by default for Group Replication channels, and it cannot be disabled for them.

For replication channels with this setting, in-memory file positions are still tracked, and file positions can still be observed for debugging purposes in error messages and through interfaces such as `SHOW REPLICA STATUS` statements (where they are shown as being invalid if they are out of date). However, the writes and reads required to persist and check the file positions are avoided in situations where GTID-based replication does not actually require them, including the transaction queuing and application process.

This option can be used only if both the replication SQL (applier) thread and replication I/O (receiver) thread are stopped. To set `GTID_ONLY = 1` for a replication channel, GTIDs must be in use on the server (`gtid_mode = ON`), and row-based binary logging must be in use on the source (statement-based replication is not supported). The options `REQUIRE_ROW_FORMAT = 1` and `SOURCE_AUTO_POSITION = 1` must be set for the replication channel.

When `GTID_ONLY = 1` is set, the replica uses `replica_parallel_workers=1` if that system variable is set to zero for the server, so it is always technically a multi-threaded applier. This is because a multi-threaded applier uses saved positions rather than the replication metadata repositories to locate the start of a transaction that it needs to reapply.

If you disable `GTID_ONLY` after setting it, the existing relay logs are deleted and the existing known binary log file positions are persisted, even if they are stale. The file positions for the binary log and relay log in the replication metadata repositories might be invalid, and a warning is returned if this is the case. Provided that `SOURCE_AUTO_POSITION` is still enabled, GTID auto-positioning is used to provide the correct positioning.

If you also disable `SOURCE_AUTO_POSITION`, the file positions for the binary log and relay log in the replication metadata repositories are used for positioning if they are valid. If they are marked as invalid, you must provide a valid binary log file name and position (`SOURCE_LOG_FILE` and `SOURCE_LOG_POS`). If you also provide a relay log file name and position (`RELAY_LOG_FILE` and `RELAY_LOG_POS`), the relay logs are preserved and the applier position is set to the stated position. GTID auto-skip ensures that any transactions already applied are skipped even if the eventual applier position is not correct.

`IGNORE_SERVER_IDS = (`*`server_id_list`*`)`

Makes the replica ignore events originating from the specified servers. The option takes a comma-separated list of 0 or more server IDs. Log rotation and deletion events from the servers are not ignored, and are recorded in the relay log.

In circular replication, the originating server normally acts as the terminator of its own events, so that they are not applied more than once. Thus, this option is useful in circular replication when one of the servers in the circle is removed. Suppose that you have a circular replication setup with 4 servers, having server IDs 1, 2, 3, and 4, and server 3 fails. When bridging the gap by starting replication from server 2 to server 4, you can include

`IGNORE_SERVER_IDS = (3)` in the `CHANGE REPLICATION SOURCE TO` statement that you issue on server 4 to tell it to use server 2 as its source instead of server 3. Doing so causes it to ignore and not to propagate any statements that originated with the server that is no longer in use.

If `IGNORE_SERVER_IDS` contains the server's own ID and the server was started with the `--replicate-same-server-id` option enabled, an error results.

Note

When global transaction identifiers (GTIDs) are used for replication, transactions that have already been applied are automatically ignored, so the `IGNORE_SERVER_IDS` function is not required and is deprecated. If `gtid_mode=ON` is set for the server, a deprecation warning is issued if you include the `IGNORE_SERVER_IDS` option in a `CHANGE REPLICATION SOURCE TO` statement.

The source metadata repository and the output of `SHOW REPLICA STATUS` provide the list of servers that are currently ignored. For more information, see Section 17.2.4.2, “Replication Metadata Repositories”, and Section 13.7.7.35, “SHOW REPLICA STATUS Statement”.

If a `CHANGE REPLICATION SOURCE TO` statement is issued without any `IGNORE_SERVER_IDS` option, any existing list is preserved. To clear the list of ignored servers, it is necessary to use the option with an empty list:

```
CHANGE REPLICATION SOURCE TO IGNORE_SERVER_IDS = ();
```

`RESET REPLICA ALL` clears `IGNORE_SERVER_IDS`.

Note

A deprecation warning is issued if `SET GTID_MODE=ON` is issued when any channel has existing server IDs set with `IGNORE_SERVER_IDS`. Before starting GTID-based replication, check for and clear all ignored server ID lists on the servers involved. The `SHOW REPLICA STATUS` statement displays the list of ignored IDs, if there is one. If you do receive the deprecation warning, you can still clear a list after `gtid_mode=ON` is set by issuing a `CHANGE REPLICATION SOURCE TO` statement containing the `IGNORE_SERVER_IDS` option with an empty list.

`NETWORK_NAMESPACE = 'namespace'`

The network namespace to use for TCP/IP connections to the replication source server or, if the MySQL communication stack is in use, for Group Replication’s group communication connections. The maximum length of the string value is 64 characters. If

this option is omitted, connections from the replica use the default (global) namespace. On platforms that do not implement network namespace support, failure occurs when the replica attempts to connect to the source. For information about network namespaces, see Section 5.1.14, “Network Namespace Support”. `NETWORK_NAMESPACE` is available as of MySQL 8.0.22.

`PRIVILEGE_CHECKS_USER = {NULL | 'account'}`

Names a user account that supplies a security context for the specified channel. `NULL`, which is the default, means no security context is used. `PRIVILEGE_CHECKS_USER` is available as of MySQL 8.0.18.

The user name and host name for the user account must follow the syntax described in Section 6.2.4, “Specifying Account Names”, and the user must not be an anonymous user (with a blank user name) or the `CURRENT_USER`. The account must have the `REPLICATION_APPLIER` privilege, plus the required privileges to execute the transactions replicated on the channel. For details of the privileges required by the account, see Section 17.3.3, “Replication Privilege Checks”. When you restart the replication channel, the privilege checks are applied from that point on. If you do not specify a channel and no other channels exist, the statement is applied to the default channel.

The use of row-based binary logging is strongly recommended when `PRIVILEGE_CHECKS_USER` is set, and you can set `REQUIRE_ROW_FORMAT` to enforce this. For example, to start privilege checks on the channel `channel_1` on a running replica, issue the following statements:

```
STOP REPLICA FOR CHANNEL 'channel_1';

CHANGE REPLICATION SOURCE TO
    PRIVILEGE_CHECKS_USER = 'user'@'host',
    REQUIRE_ROW_FORMAT = 1,
    FOR CHANNEL 'channel_1';

START REPLICA FOR CHANNEL 'channel_1';
```

`RELAY_LOG_FILE = 'relay_log_file'`, `RELAY_LOG_POS = 'relay_log_pos'`

The relay log file name, and the location in that file, at which the replication SQL thread begins reading from the replica's relay log the next time the thread starts. `RELAY_LOG_FILE` can use either an absolute or relative path, and uses the same base name as `SOURCE_LOG_FILE`. The maximum length of the string value is 511 characters.

A `CHANGE REPLICATION SOURCE TO` statement using `RELAY_LOG_FILE`, `RELAY_LOG_POS`, or both options can be executed on a running replica when the replication SQL (applier) thread is stopped. Relay logs are preserved if at least one of the replication applier thread and the replication I/O (receiver) thread is running. If both threads are stopped, all relay log files are deleted unless at least one of `RELAY_LOG_FILE` or `RELAY_LOG_POS` is specified. For the Group Replication applier channel (`group_replication_applier`), which only has an applier thread and no receiver thread, this is the case if the applier thread is stopped, but with that channel you cannot use the `RELAY_LOG_FILE` and `RELAY_LOG_POS` options.

`REQUIRE_ROW_FORMAT = {0|1}`

Permits only row-based replication events to be processed by the replication channel. This option prevents the replication applier from

taking actions such as creating temporary tables and executing `LOAD DATA INFILE` requests, which increases the security of the channel. The `REQUIRE_ROW_FORMAT` option is disabled by default for asynchronous replication channels, but it is enabled by default for Group Replication channels, and it cannot be disabled for them. For more information, see Section 17.3.3, “Replication Privilege Checks”. `REQUIRE_ROW_FORMAT` is available as of MySQL 8.0.19.

`REQUIRE_TABLE_PRIMARY_KEY_CHECK = {STREAM | ON | OFF | GENERATE}`

Available as of MySQL 8.0.20, this option lets a replica set its own policy for primary key checks, as follows:

- `ON`: The replica sets `sql_require_primary_key = ON`; any replicated `CREATE TABLE` or `ALTER TABLE` statement must result in a table that contains a primary key.
- `OFF`: The replica sets `sql_require_primary_key = OFF`; no replicated `CREATE TABLE` or `ALTER TABLE` statement is checked for the presence of a primary key.
- `STREAM`: The replica uses whatever value of `sql_require_primary_key` is replicated from the source for each transaction. This is the default value, and the default behavior.
- `GENERATE`: Added in MySQL 8.0.32, this causes the replica to generate an invisible primary key for any `InnoDB` table that, as replicated, lacks a primary key. See Section 13.1.20.11, “Generated Invisible Primary Keys”, for more information.

  `GENERATE` is not compatible with Group Replication; you can use `ON`, `OFF`, or `STREAM`.

A divergence based on the presence of a generated invisible primary key solely on a source or replica table is supported by MySQL Replication as long as the source supports GIPKs (MySQL 8.0.30 and later) and the replica uses MySQL version 8.0.32 or later. If you use GIPKs on a replica and replicate from a source using MySQL 8.0.29 or earlier, you should be aware that in this case such divergences in schema, other than the extra GIPK on the replica, are not supported and may result in replication errors.

For multisource replication, setting `REQUIRE_TABLE_PRIMARY_KEY_CHECK` to `ON` or `OFF` lets the replica normalize behavior across replication channels for different sources, and to keep a consistent setting for `sql_require_primary_key`. Using `ON` safeguards against the accidental loss of primary keys when multiple sources update the same set of tables. Using `OFF` lets sources that can manipulate primary keys to work alongside sources that cannot.

In the case of multiple replicas, when `REQUIRE_TABLE_PRIMARY_KEY_CHECK` is set to `GENERATE`, the generated invisible primary key added by a given replica is independent of any such key added on any other replica. This means that, if generated invisible primary keys are in use, the values in the generated primary key columns on different replicas

are not guaranteed to be the same. This may be an issue when failing over to such a replica.

When `PRIVILEGE_CHECKS_USER` is `NULL` (the default), the user account does not need administration level privileges to set restricted session variables. Setting this option to a value other than `NULL` means that, when `REQUIRE_TABLE_PRIMARY_KEY_CHECK` is `ON` or `OFF`, the user account does not require session administration level privileges to set restricted session variables such as `sql_require_primary_key`, avoiding the need to grant the account such privileges. (Such privileges are required when `PRIVILEGE_CHECKS_USER` is `STREAM` or `GENERATE`.) For more information, see Section 17.3.3, “Replication Privilege Checks”.

`SOURCE_AUTO_POSITION = {0|1}`

Makes the replica attempt to connect to the source using the auto-positioning feature of GTID-based replication, rather than a binary log file based position. This option is used to start a replica using GTID-based replication. The default is 0, meaning that GTID auto-positioning and GTID-based replication are not used. This option can be used with `CHANGE REPLICATION SOURCE TO` only if both the replication SQL (applier) thread and replication I/O (receiver) thread are stopped.

Both the replica and the source must have GTIDs enabled (`GTID_MODE=ON`, `ON_PERMISSIVE`, or `OFF_PERMISSIVE` on the replica, and `GTID_MODE=ON` on the source). `SOURCE_LOG_FILE`, `SOURCE_LOG_POS`, `RELAY_LOG_FILE`, and `RELAY_LOG_POS` cannot be specified together with `SOURCE_AUTO_POSITION = 1`. If multi-source replication is enabled on the replica, you need to set the `SOURCE_AUTO_POSITION = 1` option for each applicable replication channel.

With `SOURCE_AUTO_POSITION = 1` set, in the initial connection handshake, the replica sends a GTID set containing the transactions that it has already received, committed, or both. The source responds by sending all transactions recorded in its binary log whose GTID is not included in the GTID set sent by the replica. This exchange ensures that the source only sends the transactions with a GTID that the replica has not already recorded or committed. If the replica receives transactions from more than one source, as in the case of a diamond topology, the auto-skip function ensures that the transactions are not applied twice. For details of how the GTID set sent by the replica is computed, see Section 17.1.3.3, “GTID Auto-Positioning”.

If any of the transactions that should be sent by the source have been purged from the source's binary log, or added to the set of GTIDs in the `gtid_purged` system variable by another method, the source sends the error `ER_MASTER_HAS_PURGED_REQUIRED_GTIDS` to the replica, and replication does not start. The GTIDs of the missing purged transactions are identified and listed in the source's error log in the warning message `ER_FOUND_MISSING_GTIDS`. Also, if during the exchange of transactions it is found that the replica has recorded or committed transactions with the source's UUID in the GTID, but the source itself has not committed them, the source sends the error `ER_SLAVE_HAS_MORE_GTIDS_THAN_MASTER` to the replica and replication does not start. For information on how to handle these situations, see Section 17.1.3.3, “GTID Auto-Positioning”.

You can see whether replication is running with GTID auto-positioning enabled by checking the Performance Schema `replication_connection_status` table or the output of `SHOW REPLICA STATUS`. Disabling the `SOURCE_AUTO_POSITION` option again makes the replica revert to file-based replication.

`SOURCE_BIND = 'interface_name'`

Determines which of the replica's network interfaces is chosen for connecting to the source, for use on replicas that have multiple network interfaces. Specify the IP address of the network interface. The maximum length of the string value is 255 characters.

The IP address configured with this option, if any, can be seen in the `Source_Bind` column of the output from `SHOW REPLICA STATUS`. In the source metadata repository table `mysql.slave_master_info`, the value can be seen as the `Source_bind` column. The ability to bind a replica to a specific network interface is also supported by NDB Cluster.

`SOURCE_COMPRESSION_ALGORITHMS = 'algorithm[,algorithm][,algorithm]'`

Specifies one, two, or three of the permitted compression algorithms for connections to the replication source server, separated by commas. The maximum length of the string value is 99 characters. The default value is `uncompressed`.

The available algorithms are `zlib`, `zstd`, and `uncompressed`, the same as for the `protocol_compression_algorithms` system variable. The algorithms can be specified in any order, but it is not an order of preference - the algorithm negotiation process attempts to use `zlib`, then `zstd`, then `uncompressed`, if they are specified. `SOURCE_COMPRESSION_ALGORITHMS` is available as of MySQL 8.0.18.

The value of `SOURCE_COMPRESSION_ALGORITHMS` applies only if the `replica_compressed_protocol` or `slave_compressed_protocol` system variable is disabled. If `replica_compressed_protocol` or `slave_compressed_protocol` is enabled, it takes precedence over `SOURCE_COMPRESSION_ALGORITHMS` and connections to the source use `zlib` compression if both source and replica support that algorithm. For more information, see Section 4.2.8, “Connection Compression Control”.

Binary log transaction compression (available as of MySQL 8.0.20), which is activated by the `binlog_transaction_compression` system variable, can also be used to save bandwidth. If you do this in combination with connection compression, connection compression has less opportunity to act on the data, but can still compress headers and those events and transaction payloads that are uncompressed. For more information on binary log transaction compression, see Section 5.4.4.5, “Binary Log Transaction Compression”.

`SOURCE_CONNECT_RETRY = interval`

Specifies the interval in seconds between the reconnection attempts that the replica makes after the connection to the source times out. The default interval is 60.

The number of attempts is limited by the `SOURCE_RETRY_COUNT` option. If both the default settings are used, the replica waits 60 seconds between reconnection attempts (`SOURCE_CONNECT_RETRY=60`), and keeps attempting to

reconnect at this rate for 60 days (`SOURCE_RETRY_COUNT=86400`). These values are recorded in the source metadata repository and shown in the `replication_connection_configuration` Performance Schema table.

`SOURCE_CONNECTION_AUTO_FAILOVER = {0|1}`

Activates the asynchronous connection failover mechanism for a replication channel if one or more alternative replication source servers are available (so when there are multiple MySQL servers or groups of servers that share the replicated data). `SOURCE_CONNECTION_AUTO_FAILOVER` is available as of MySQL 8.0.22. The default is 0, meaning that the mechanism is not activated. For full information and instructions to set up this feature, see Section 17.4.9.2, “Asynchronous Connection Failover for Replicas”.

The asynchronous connection failover mechanism takes over after the reconnection attempts controlled by `SOURCE_CONNECT_RETRY` and `SOURCE_RETRY_COUNT` are exhausted. It reconnects the replica to an alternative source chosen from a specified source list, which you manage using the `asynchronous_connection_failover_add_source` and `asynchronous_connection_failover_delete_source` functions. To add and remove managed groups of servers, use the `asynchronous_connection_failover_add_managed` and `asynchronous_connection_failover_delete_managed` functions instead. For more information, see Section 17.4.9, “Switching Sources and Replicas with Asynchronous Connection Failover”.

Important

1. You can only set `SOURCE_CONNECTION_AUTO_FAILOVER = 1` when GTID auto-positioning is in use (`SOURCE_AUTO_POSITION = 1`).

2. When you set `SOURCE_CONNECTION_AUTO_FAILOVER = 1`, set `SOURCE_RETRY_COUNT` and `SOURCE_CONNECT_RETRY` to minimal numbers that just allow a few retry attempts with the same source, in case the connection failure is caused by a transient network outage. Otherwise the asynchronous connection failover mechanism cannot be activated promptly. Suitable values are `SOURCE_RETRY_COUNT=3` and `SOURCE_CONNECT_RETRY=10`, which make the replica retry the connection 3 times with 10-second intervals between.

3. When you set `SOURCE_CONNECTION_AUTO_FAILOVER = 1`, the replication metadata repositories must contain the credentials for a replication user account that can be used to connect to all the servers on the source list for the replication channel.

The account must also have `SELECT` permissions on the Performance Schema tables. These credentials can be set using the `CHANGE REPLICATION SOURCE TO` statement with the `SOURCE_USER` and `SOURCE_PASSWORD` options. For more information, see Section 17.4.9, "Switching Sources and Replicas with Asynchronous Connection Failover".

4. From MySQL 8.0.27, when you set `SOURCE_CONNECTION_AUTO_FAILOVER = 1`, asynchronous connection failover for replicas is automatically activated if this replication channel is on a Group Replication primary in a group in single-primary mode. With this function active, if the primary that is replicating goes offline or into an error state, the new primary starts replication on the same channel when it is elected. If you want to use the function, this replication channel must also be set up on all the secondary servers in the replication group, and on any new joining members. (If the servers are provisioned using MySQL's clone functionality, this all happens automatically.) If you do not want to use the function, disable it by using the `group_replication_disable_member_action` function to disable the Group Replication member action `mysql_start_failover_channels_if_primary` which is enabled by default. For more information, see Section 17.4.9.2, "Asynchronous Connection Failover for Replicas".

`SOURCE_DELAY = interval`

Specifies how many seconds behind the source the replica must lag. An event received from the source is not executed until at least *`interval`* seconds later than its execution on the source. *`interval`* must be a nonnegative integer in the range from 0 to $2^{31}-1$. The default is 0. For more information, see Section 17.4.11, "Delayed Replication".

A `CHANGE REPLICATION SOURCE TO` statement using the `SOURCE_DELAY` option can be executed on a running replica when the replication SQL thread is stopped.

`SOURCE_HEARTBEAT_PERIOD = interval`

Controls the heartbeat interval, which stops the connection timeout occurring in the absence of data if the connection is still good. A heartbeat signal is sent to the replica after that number of seconds, and the waiting period is reset whenever the source's binary log is updated with an event. Heartbeats are therefore sent by the source

only if there are no unsent events in the binary log file for a period longer than this.

The heartbeat interval *`interval`* is a decimal value having the range 0 to 4294967 seconds and a resolution in milliseconds; the smallest nonzero value is 0.001. Setting *`interval`* to 0 disables heartbeats altogether. The heartbeat interval defaults to half the value of the `replica_net_timeout` or `slave_net_timeout` system variable. It is recorded in the source metadata repository and shown in the `replication_connection_configuration` Performance Schema table.

The system variable `replica_net_timeout` (from MySQL 8.0.26) or `slave_net_timeout` (before MySQL 8.0.26) specifies the number of seconds that the replica waits for either more data or a heartbeat signal from the source, before the replica considers the connection broken, aborts the read, and tries to reconnect. The default value is 60 seconds (one minute). Note that a change to the value or default setting of `replica_net_timeout` or `slave_net_timeout` does not automatically change the heartbeat interval, whether that has been set explicitly or is using a previously calculated default. A warning is issued if you set the global value of `replica_net_timeout` or `slave_net_timeout` to a value less than that of the current heartbeat interval. If `replica_net_timeout` or `slave_net_timeout` is changed, you must also issue `CHANGE REPLICATION SOURCE TO` to adjust the heartbeat interval to an appropriate value so that the heartbeat signal occurs before the connection timeout. If you do not do this, the heartbeat signal has no effect, and if no data is received from the source, the replica can make repeated reconnection attempts, creating zombie dump threads.

`SOURCE_HOST = '`*`host_name`*`'`

The host name or IP address of the replication source server. The replica uses this to connect to the source. The maximum length of the string value is 255 characters.

If you specify `SOURCE_HOST` or `SOURCE_PORT`, the replica assumes that the source server is different from before (even if the option value is the same as its current value.) In this case, the old values for the source's binary log file name and position are considered no longer applicable, so if you do not specify `SOURCE_LOG_FILE` and `SOURCE_LOG_POS` in the statement, `SOURCE_LOG_FILE=''` and `SOURCE_LOG_POS=4` are silently appended to it.

Setting `SOURCE_HOST=''` (that is, setting its value explicitly to an empty string) is *not* the same as not setting `SOURCE_HOST` at all. Trying to set `SOURCE_HOST` to an empty string fails with an error.

`SOURCE_LOG_FILE = '`*`source_log_name`*`', SOURCE_LOG_POS =` *`source_log_pos`*

The binary log file name, and the location in that file, at which the replication I/O (receiver) thread begins reading from the source's binary log the next time the thread starts. Specify these options if you are using binary log file position based replication.

`SOURCE_LOG_FILE` must include the numeric suffix of a specific binary log file that is available on the source server, for example,

`SOURCE_LOG_FILE='binlog.000145'`. The maximum length of the string value is 511 characters.

`SOURCE_LOG_POS` is the numeric position for the replica to start reading in that file. `SOURCE_LOG_POS=4` represents the start of the events in a binary log file.

If you specify either of `SOURCE_LOG_FILE` or `SOURCE_LOG_POS`, you cannot specify `SOURCE_AUTO_POSITION = 1`, which is for GTID-based replication.

If neither of `SOURCE_LOG_FILE` or `SOURCE_LOG_POS` is specified, the replica uses the last coordinates of the *replication SQL thread* before `CHANGE REPLICATION SOURCE TO` was issued. This ensures that there is no discontinuity in replication, even if the replication SQL (applier) thread was late compared to the replication I/O (receiver) thread.

`SOURCE_PASSWORD = 'password'`

The password for the replication user account to use for connecting to the replication source server. The maximum length of the string value is 32 characters. If you specify `SOURCE_PASSWORD`, `SOURCE_USER` is also required.

The password used for a replication user account in a `CHANGE REPLICATION SOURCE TO` statement is limited to 32 characters in length. Trying to use a password of more than 32 characters causes `CHANGE REPLICATION SOURCE TO` to fail.

The password is masked in MySQL Server's logs, Performance Schema tables, and `SHOW PROCESSLIST` statements.

`SOURCE_PORT = port_num`

The TCP/IP port number that the replica uses to connect to the replication source server.

Note

Replication cannot use Unix socket files. You must be able to connect to the replication source server using TCP/IP.

If you specify `SOURCE_HOST` or `SOURCE_PORT`, the replica assumes that the source server is different from before (even if the option value is the same as its current value.) In this case, the old values for the source's binary log file name and position are considered no longer applicable, so if you do not specify `SOURCE_LOG_FILE` and `SOURCE_LOG_POS` in the statement, `SOURCE_LOG_FILE=''` and `SOURCE_LOG_POS=4` are silently appended to it.

`SOURCE_PUBLIC_KEY_PATH = 'key_file_name'`

Enables RSA key pair-based password exchange by providing the path name to a file containing a replica-side copy of the public key required by the source. The file must be in PEM format. The maximum length of the string value is 511 characters.

This option applies to replicas that authenticate with the `sha256_password` or `caching_sha2_password` authentication plugin. (For `sha256_password`, `SOURCE_PUBLIC_KEY_PATH` can be used only if MySQL was built using OpenSSL.) If you are using a replication user account that authenticates with the `caching_sha2_password` plugin (which is the default from MySQL 8.0), and you are not using a secure connection, you must

specify either this option or the `GET_SOURCE_PUBLIC_KEY=1` option to provide the RSA public key to the replica.

`SOURCE_RETRY_COUNT = count`

Sets the maximum number of reconnection attempts that the replica makes after the connection to the source times out, as determined by the `replica_net_timeout` or `slave_net_timeout` system variable. If the replica does need to reconnect, the first retry occurs immediately after the timeout. The default is 86400 attempts.

The interval between the attempts is specified by the `SOURCE_CONNECT_RETRY` option. If both the default settings are used, the replica waits 60 seconds between reconnection attempts (`SOURCE_CONNECT_RETRY=60`), and keeps attempting to reconnect at this rate for 60 days (`SOURCE_RETRY_COUNT=86400`). A setting of 0 for `SOURCE_RETRY_COUNT` means that there is no limit on the number of reconnection attempts, so the replica keeps trying to reconnect indefinitely.

The values for `SOURCE_CONNECT_RETRY` and `SOURCE_RETRY_COUNT` are recorded in the source metadata repository and shown in the `replication_connection_configuration` Performance Schema table. `SOURCE_RETRY_COUNT` supersedes the `--master-retry-count` server startup option.

`SOURCE_SSL = {0|1}`

Specify whether the replica encrypts the replication connection. The default is 0, meaning that the replica does not encrypt the replication connection. If you set `SOURCE_SSL=1`, you can configure the encryption using the `SOURCE_SSL_xxx` and `SOURCE_TLS_xxx` options.

Setting `SOURCE_SSL=1` for a replication connection and then setting no further `SOURCE_SSL_xxx` options corresponds to setting `--ssl-mode=REQUIRED` for the client, as described in Command Options for Encrypted Connections. With `SOURCE_SSL=1`, the connection attempt only succeeds if an encrypted connection can be established. A replication connection does not fall back to an unencrypted connection, so there is no setting corresponding to the `--ssl-mode=PREFERRED` setting for replication. If `SOURCE_SSL=0` is set, this corresponds to `--ssl-mode=DISABLED`.

Important

To help prevent sophisticated man-in-the-middle attacks, it is important for the replica to verify the server's identity. You can specify additional `SOURCE_SSL_xxx` options to correspond to the settings `--ssl-mode=VERIFY_CA` and `--ssl-mode=VERIFY_IDENTITY`, which are a better choice than the default setting to help prevent this type of attack. With these settings, the replica checks that the server's certificate is valid, and checks that the host name the replica is using matches the identity in the server's certificate. To implement one of these levels of verification, you must first ensure that the CA certificate for the server is reliably available to the

replica, otherwise availability issues will result. For this reason, they are not the default setting.

`SOURCE_SSL_xxx`, `SOURCE_TLS_xxx`

Specify how the replica uses encryption and ciphers to secure the replication connection. These options can be changed even on replicas that are compiled without SSL support. They are saved to the source metadata repository, but are ignored if the replica does not have SSL support enabled. The maximum length of the value for the string-valued `SOURCE_SSL_xxx` and `SOURCE_TLS_xxx` options is 511 characters, with the exception of `SOURCE_TLS_CIPHERSUITES`, for which it is 4000 characters.

The `SOURCE_SSL_xxx` and `SOURCE_TLS_xxx` options perform the same functions as the `--ssl-xxx` and `--tls-xxx` client options described in Command Options for Encrypted Connections. The correspondence between the two sets of options, and the use of the `SOURCE_SSL_xxx` and `SOURCE_TLS_xxx` options to set up a secure connection, is explained in Section 17.3.1, “Setting Up Replication to Use Encrypted Connections”.

`SOURCE_USER = 'user_name'`

The user name for the replication user account to use for connecting to the replication source server. The maximum length of the string value is 96 characters.

For Group Replication, this account must exist on every member of the replication group. It is used for distributed recovery if the XCom communication stack is in use for the group, and also used for group communication connections if the MySQL communication stack is in use for the group. With the MySQL communication stack, the account must have the `GROUP_REPLICATION_STREAM` permission.

It is possible to set an empty user name by specifying `SOURCE_USER=''`, but the replication channel cannot be started with an empty user name. In releases before MySQL 8.0.21, only set an empty `SOURCE_USER` user name if you need to clear previously used credentials from the replication metadata repositories for security purposes. Do not use the channel afterwards, due to a bug in these releases that can substitute a default user name if an empty user name is read from the repositories (for example, during an automatic restart of a Group Replication channel). From MySQL 8.0.21, it is valid to set an empty `SOURCE_USER` user name and use the channel afterwards if you always provide user credentials using the `START REPLICA` statement or `START GROUP_REPLICATION` statement that starts the replication channel. This approach means that the replication channel always needs operator intervention to restart, but the user credentials are not recorded in the replication metadata repositories.

Important

To connect to the source using a replication user account that authenticates with the `caching_sha2_password` plugin, you must either set up a secure connection as described in Section 17.3.1, “Setting Up Replication to Use Encrypted Connections”, or enable the unencrypted connection to support password exchange using an RSA

key pair. The `caching_sha2_password` authentication plugin is the default for new users created from MySQL 8.0 (for details, see Section 6.4.1.2, “Caching SHA-2 Pluggable Authentication”). If the user account that you create or use for replication uses this authentication plugin, and you are not using a secure connection, you must enable RSA key pair-based password exchange for a successful connection. You can do this using either the `SOURCE_PUBLIC_KEY_PATH` option or the `GET_SOURCE_PUBLIC_KEY=1` option for this statement.

`SOURCE_ZSTD_COMPRESSION_LEVEL = level`

The compression level to use for connections to the replication source server that use the `zstd` compression algorithm. `SOURCE_ZSTD_COMPRESSION_LEVEL` is available as of MySQL 8.0.18. The permitted levels are from 1 to 22, with larger values indicating increasing levels of compression. The default level is 3.

The compression level setting has no effect on connections that do not use `zstd` compression. For more information, see Section 4.2.8, “Connection Compression Control”.

Examples

`CHANGE REPLICATION SOURCE TO` is useful for setting up a replica when you have the snapshot of the source and have recorded the source's binary log coordinates corresponding to the time of the snapshot. After loading the snapshot into the replica to synchronize it with the source, you can run `CHANGE REPLICATION SOURCE TO SOURCE_LOG_FILE='log_name', SOURCE_LOG_POS=log_pos` on the replica to specify the coordinates at which the replica should begin reading the source's binary log. The following example changes the source server the replica uses and establishes the source's binary log coordinates from which the replica begins reading:

```
CHANGE REPLICATION SOURCE TO
  SOURCE_HOST='source2.example.com',
  SOURCE_USER='replication',
  SOURCE_PASSWORD='password',
  SOURCE_PORT=3306,
  SOURCE_LOG_FILE='source2-bin.001',
  SOURCE_LOG_POS=4,
  SOURCE_CONNECT_RETRY=10;
```

For the procedure to switch an existing replica to a new source during failover, see Section 17.4.8, “Switching Sources During Failover”.

When GTIDs are in use on the source and the replica, specify GTID auto-positioning instead of giving the binary log file position, as in the following example. For full instructions to configure and start GTID-based replication on new or stopped servers, online servers, or additional replicas, see Section 17.1.3, “Replication with Global Transaction Identifiers”.

```
CHANGE REPLICATION SOURCE TO
  SOURCE_HOST='source3.example.com',
  SOURCE_USER='replication',
  SOURCE_PASSWORD='password',
  SOURCE_PORT=3306,
  SOURCE_AUTO_POSITION = 1,
  FOR CHANNEL "source_3";
```

In this example, multi-source replication is in use, and the `CHANGE REPLICATION SOURCE TO` statement is applied to the replication channel `"source_3"` that connects the replica to the specified

host. For guidance on setting up multi-source replication, see Section 17.1.5, “MySQL Multi-Source Replication”.

The next example shows how to make the replica apply transactions from relay log files that you want to repeat. To do this, the source need not be reachable. You can use `CHANGE REPLICATION SOURCE TO` to locate the relay log position where you want the replica to start reapplying transactions, and then start the SQL thread:

```
CHANGE REPLICATION SOURCE TO
  RELAY_LOG_FILE='replica-relay-bin.006',
  RELAY_LOG_POS=4025;
START REPLICA SQL_THREAD;
```

`CHANGE REPLICATION SOURCE TO` can also be used to skip over transactions in the binary log that are causing replication to stop. The appropriate method to do this depends on whether GTIDs are in use or not. For instructions to skip transactions using `CHANGE REPLICATION SOURCE TO` or another method, see Section 17.1.7.3, “Skipping Transactions”.

13.4.2.4 MASTER_POS_WAIT() Statement

```
SELECT MASTER_POS_WAIT('source_log_file', source_log_pos [, timeout][, channel])
```

This is actually a function, not a statement. It is used to ensure that the replica has read and executed events up to a given position in the source's binary log. See Section 12.24, “Miscellaneous Functions”, for a full description.

From MySQL 8.0.26, `MASTER_POS_WAIT` is deprecated and the alias `SOURCE_POS_WAIT` should be used instead. In releases before MySQL 8.0.26, use `MASTER_POS_WAIT`.

13.4.2.5 RESET REPLICA Statement

```
RESET REPLICA [ALL] [channel_option]

channel_option:
    FOR CHANNEL channel
```

`RESET REPLICA` makes the replica forget its position in the source's binary log. From MySQL 8.0.22, use `RESET REPLICA` in place of `RESET SLAVE`, which is deprecated from that release. In releases before MySQL 8.0.22, use `RESET SLAVE`.

This statement is meant to be used for a clean start; it clears the replication metadata repositories, deletes all the relay log files, and starts a new relay log file. It also resets to 0 the replication delay specified with the `SOURCE_DELAY` | `MASTER_DELAY` option of the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23).

Note

All relay log files are deleted, even if they have not been completely executed by the replication SQL thread. (This is a condition likely to exist on a replica if you have issued a `STOP REPLICA` statement or if the replica is highly loaded.)

For a server where GTIDs are in use (`gtid_mode` is `ON`), issuing `RESET REPLICA` has no effect on the GTID execution history. The statement does not change the values of `gtid_executed` or `gtid_purged`, or the `mysql.gtid_executed` table. If you need to reset the GTID execution history, use `RESET MASTER`, even if the GTID-enabled server is a replica where binary logging is disabled.

`RESET REPLICA` requires the `RELOAD` privilege.

To use `RESET REPLICA`, the replication SQL thread and replication I/O (receiver) thread must be stopped, so on a running replica use `STOP REPLICA` before issuing `RESET REPLICA`. To use `RESET REPLICA` on a Group Replication group member, the member status must be `OFFLINE`, meaning that the plugin is loaded but the member does not currently belong to any group. A group member can be taken offline by using a `STOP GROUP REPLICATION` statement.

The optional `FOR CHANNEL channel` clause enables you to name which replication channel the statement applies to. Providing a `FOR CHANNEL channel` clause applies the `RESET REPLICA` statement to a specific replication channel. Combining a `FOR CHANNEL channel` clause with the `ALL` option deletes the specified channel. If no channel is named and no extra channels exist, the statement applies to the default channel. Issuing a `RESET REPLICA ALL` statement without a `FOR CHANNEL channel` clause when multiple replication channels exist deletes *all* replication channels and recreates only the default channel. See Section 17.2.2, “Replication Channels” for more information.

`RESET REPLICA` does not change any replication connection parameters, which include the source's host name and port, the replication user account and its password, the `PRIVILEGE_CHECKS_USER` account, the `REQUIRE_ROW_FORMAT` option, the `REQUIRE_TABLE_PRIMARY_KEY_CHECK` option,and the `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` option. If you want to change any of the replication connection parameters, you can do this using a `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23) after the server start. If you want to remove all of the replication connection parameters, use `RESET REPLICA ALL`. `RESET REPLICA ALL` also clears the `IGNORE_SERVER_IDS` list set by `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO`. When you have used `RESET REPLICA ALL`, if you want to use the instance as a replica again, you need to issue a `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement after the server start to specify new connection parameters.

From MySQL 8.0.27, you can set the `GTID_ONLY` option on the `CHANGE REPLICATION SOURCE TO` statement to stop a replication channel from persisting file names and file positions in the replication metadata repositories. When you issue a `RESET REPLICA` statement, the replication metadata repositories are synchronized. `RESET REPLICA ALL` deletes rather than updates the repositories, so they are synchronized implicitly.

In the event of an unexpected server exit or deliberate restart after issuing `RESET REPLICA` but before issuing `START REPLICA`, retention of the replication connection parameters depends on the repository used for the replication metadata:

- When `master_info_repository=TABLE` and `relay_log_info_repository=TABLE` are set on the server (which are the default settings from MySQL 8.0), replication connection parameters are preserved in the crash-safe `InnoDB` tables `mysql.slave_master_info` and `mysql.slave_relay_log_info` as part of the `RESET REPLICA` operation. They are also retained in memory. In the event of an unexpected server exit or deliberate restart after issuing `RESET REPLICA` but before issuing `START REPLICA`, the replication connection parameters are retrieved from the tables and reapplied to the channel. This situation applies from MySQL 8.0.13 for the connection metadata repository, and from MySQL 8.0.19 for the applier metadata repository.
- If `master_info_repository=FILE` and `relay_log_info_repository=FILE` are set on the server, which is deprecated from MySQL 8.0, or the MySQL Server release is earlier than those specified above, replication connection parameters are only retained in memory. If the replica `mysqld` is restarted immediately after issuing `RESET REPLICA` due to an unexpected server exit or deliberate restart, the connection parameters are lost. In that case, you must issue a `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23) after the server start to respecify the connection parameters before issuing `START REPLICA`.

`RESET REPLICA` does not change any replication filter settings (such as `--replicate-ignore-table`) for channels affected by the statement. However, `RESET REPLICA ALL` removes the replication filters that were set on the channels deleted by the statement. When the deleted channel or channels are recreated, any global replication filters specified for the replica are copied to them, and no channel specific replication filters are applied. For more information see Section 17.2.5.4, “Replication Channel Based Filters”.

`RESET REPLICA` causes an implicit commit of an ongoing transaction. See Section 13.3.3, “Statements That Cause an Implicit Commit”.

If the replication SQL thread was in the middle of replicating temporary tables when it was stopped, and `RESET REPLICA` is issued, these replicated temporary tables are deleted on the replica.

Note

When used on an NDB Cluster replica SQL node, `RESET REPLICA` clears the `mysql.ndb_apply_status` table. You should keep in mind when using this statement that `ndb_apply_status` uses the `NDB` storage engine and so is shared by all SQL nodes attached to the cluster.

You can override this behavior by issuing `SET GLOBAL @@ndb_clear_apply_status=OFF` prior to executing `RESET REPLICA`, which keeps the replica from purging the `ndb_apply_status` table in such cases.

13.4.2.6 RESET SLAVE Statement

```
RESET {SLAVE | REPLICA} [ALL] [channel_option]

channel_option:
    FOR CHANNEL channel
```

Makes the replica forget its position in the source's binary log. From MySQL 8.0.22, `RESET SLAVE` is deprecated and the alias `RESET REPLICA` should be used instead. In releases before MySQL 8.0.22, use `RESET SLAVE`. The statement works in the same way as before, only the terminology used for the statement and its output has changed. Both versions of the statement update the same status variables when used. Please see the documentation for `RESET REPLICA` for a description of the statement.

13.4.2.7 SOURCE_POS_WAIT() Statement

```
SELECT SOURCE_POS_WAIT('source_log_file', source_log_pos [, timeout][, channel])
```

This is actually a function, not a statement. It is used to ensure that the replica has read and executed events up to a given position in the source's binary log. See Section 12.24, “Miscellaneous Functions”, for a full description.

From MySQL 8.0.26, use `SOURCE_POS_WAIT` in place of `MASTER_POS_WAIT`, which is deprecated from that release. In releases before MySQL 8.0.26, use `MASTER_POS_WAIT`.

13.4.2.8 START REPLICA Statement

```
START REPLICA [thread_types] [until_option] [connection_options] [channel_option]

thread_types:
    [thread_type [, thread_type] ... ]

thread_type:
    IO_THREAD | SQL_THREAD

until_option:
    UNTIL {   {SQL_BEFORE_GTIDS | SQL_AFTER_GTIDS} = gtid_set
          |   MASTER_LOG_FILE = 'log_name', MASTER_LOG_POS = log_pos
          |   SOURCE_LOG_FILE = 'log_name', SOURCE_LOG_POS = log_pos
          |   RELAY_LOG_FILE = 'log_name', RELAY_LOG_POS = log_pos
          |   SQL_AFTER_MTS_GAPS  }

connection_options:
    [USER='user_name'] [PASSWORD='user_pass'] [DEFAULT_AUTH='plugin_name'] [PLUGIN_DIR='plugin_dir']


channel_option:
    FOR CHANNEL channel

gtid_set:
    uuid_set [, uuid_set] ...
    | ''

uuid_set:
```

```
    uuid:interval[:interval]...

uuid:
    hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh

h:
    [0-9,A-F]

interval:
    n[-n]

    (n >= 1)
```

`START REPLICA` starts the replication threads, either together or separately. From MySQL 8.0.22, use `START REPLICA` in place of `START SLAVE`, which is deprecated from that release. In releases before MySQL 8.0.22, use `START SLAVE`.

`START REPLICA` requires the `REPLICATION_SLAVE_ADMIN` privilege (or the deprecated `SUPER` privilege). `START REPLICA` causes an implicit commit of an ongoing transaction. See Section 13.3.3, "Statements That Cause an Implicit Commit".

For the thread type options, you can specify `IO_THREAD`, `SQL_THREAD`, both of these, or neither of them. Only the threads that are started are affected by the statement.

- `START REPLICA` with no thread type options starts all of the replication threads, and so does `START REPLICA` with both of the thread type options.
- `IO_THREAD` starts the replication receiver thread, which reads events from the source server and stores them in the relay log.
- `SQL_THREAD` starts the replication applier thread, which reads events from the relay log and executes them. A multithreaded replica (with `replica_parallel_workers` or `slave_parallel_workers` > 0) applies transactions using a coordinator thread and multiple applier threads, and `SQL_THREAD` starts all of these.

Important

`START REPLICA` sends an acknowledgment to the user after all the replication threads have started. However, the replication receiver thread might not yet have connected to the source successfully, or an applier thread might stop when applying an event right after starting. `START REPLICA` does not continue to monitor the threads after they are started, so it does not warn you if they subsequently stop or cannot connect. You must check the replica's error log for error messages generated by the replication threads, or check that they are running satisfactorily with `SHOW REPLICA STATUS`. A successful `START REPLICA` statement causes `SHOW REPLICA STATUS` to show `Replica_SQL_Running=Yes`, but it might or might not show `Replica_IO_Running=Yes`, because `Replica_IO_Running=Yes` is only shown if the receiver thread is both running and connected. For more information, see Section 17.1.7.1, "Checking Replication Status".

The optional `FOR CHANNEL channel` clause enables you to name which replication channel the statement applies to. Providing a `FOR CHANNEL channel` clause applies the `START REPLICA` statement to a specific replication channel. If no clause is named and no extra channels exist, the statement applies to the default channel. If a `START REPLICA` statement does not have a channel defined when using multiple channels, this statement starts the specified threads for all channels. See Section 17.2.2, "Replication Channels" for more information.

The replication channels for Group Replication (`group_replication_applier` and `group_replication_recovery`) are managed automatically by the server instance. `START REPLICA` cannot be used at all with the `group_replication_recovery` channel, and should only be used with the `group_replication_applier` channel when Group Replication is not running.

The `group_replication_applier` channel only has an applier thread and has no receiver thread, so it can be started if required by using the `SQL_THREAD` option without the `IO_THREAD` option.

`START REPLICA` supports pluggable user-password authentication (see Section 6.2.17, "Pluggable Authentication") with the `USER`, `PASSWORD`, `DEFAULT_AUTH` and `PLUGIN_DIR` options, as described in the following list. When you use these options, you must start the receiver thread (`IO_THREAD` option) or all the replication threads - you cannot start the replication applier thread (`SQL_THREAD` option) alone.

`USER` — The user name for the account. You must set this if `PASSWORD` is used. The option cannot be set to an empty or null string.

`PASSWORD` — The password for the named user account.

`DEFAULT_AUTH` — The name of the authentication plugin. The default is MySQL native authentication.

`PLUGIN_DIR` — The location of the authentication plugin.

Important

The password that you set using `START REPLICA` is masked when it is written to MySQL Server's logs, Performance Schema tables, and `SHOW PROCESSLIST` statements. However, it is sent in plain text over the connection to the replica server instance. To protect the password in transit, use SSL/TLS encryption, an SSH tunnel, or another method of protecting the connection from unauthorized viewing, for the connection between the replica server instance and the client that you use to issue `START REPLICA`.

The `UNTIL` clause makes the replica start replication, then process transactions up to the point that you specify in the `UNTIL` clause, then stop again. The `UNTIL` clause can be used to make a replica proceed until just before the point where you want to skip a transaction that is unwanted, and then skip the transaction as described in Section 17.1.7.3, "Skipping Transactions". To identify a transaction, you can use `mysqlbinlog` with the source's binary log or the replica's relay log, or use a `SHOW BINLOG EVENTS` statement.

You can also use the `UNTIL` clause for debugging replication by processing transactions one at a time or in sections. If you are using the `UNTIL` clause to do this, start the replica with the `--skip-slave-start` option, or from MySQL 8.0.24, the `skip_slave_start` system variable, to prevent the SQL thread from running when the replica server starts. Remove the option or system variable setting after the procedure is complete, so that it is not forgotten in the event of an unexpected server restart.

The `SHOW REPLICA STATUS` statement includes output fields that display the current values of the `UNTIL` condition. The `UNTIL` condition lasts for as long as the affected threads are still running, and is removed when they stop.

The `UNTIL` clause operates on the replication applier thread (`SQL_THREAD` option). You can use the `SQL_THREAD` option or let the replica default to starting both threads. If you use the `IO_THREAD` option alone, the `UNTIL` clause is ignored because the applier thread is not started.

The point that you specify in the `UNTIL` clause can be any one (and only one) of the following options:

`SOURCE_LOG_FILE` and `SOURCE_LOG_POS` (from MySQL 8.0.23), or `MASTER_LOG_FILE` and `MASTER_LOG_POS` (to MySQL 8.0.22) — These options make the replication applier process transactions up to a position in its relay log, identified by the file name and file position of the corresponding point in the binary log on the source server. The applier thread finds the nearest transaction boundary at or after the specified position, finishes applying the transaction, and stops there. For compressed transaction payloads, specify the end position of the compressed `Transaction_payload_event`.

These options can still be used when the `GTID_ONLY` option was set on the `CHANGE REPLICATION SOURCE TO` statement to stop

	the replication channel from persisting file names and file positions in the replication metadata repositories. The file names and file positions are tracked in memory.
`RELAY_LOG_FILE` and `RELAY_LOG_POS`	These options make the replication applier process transactions up to a position in the replica's relay log, identified by the relay log file name and a position in that file. The applier thread finds the nearest transaction boundary at or after the specified position, finishes applying the transaction, and stops there. For compressed transaction payloads, specify the end position of the compressed `Transaction_payload_event`. These options can still be used when the `GTID_ONLY` option was set on the `CHANGE REPLICATION SOURCE TO` statement to stop the replication channel from persisting file names and file positions in the replication metadata repositories. The file names and file positions are tracked in memory.
`SQL_BEFORE_GTIDS`	This option makes the replication applier start processing transactions and stop when it encounters any transaction that is in the specified GTID set. The encountered transaction from the GTID set is not applied, and nor are any of the other transactions in the GTID set. The option takes a GTID set containing one or more global transaction identifiers as an argument (see GTID Sets). Transactions in a GTID set do not necessarily appear in the replication stream in the order of their GTIDs, so the transaction before which the applier stops is not necessarily the earliest.
`SQL_AFTER_GTIDS`	This option makes the replication applier start processing transactions and stop when it has processed all of the transactions in a specified GTID set. The option takes a GTID set containing one or more global transaction identifiers as an argument (see GTID Sets). With `SQL_AFTER_GTIDS`, the replication threads stop when they have processed the transactions in the GTID set and encounter the first transaction whose GTID is not part of the GTID set. For example, executing `START REPLICA UNTIL SQL_AFTER_GTIDS = 3E11FA47-71CA-11E1-9E33-C80AA9429562:11-56` causes the replica to obtain all transactions just mentioned from the source, including all of the transactions having the sequence numbers 11 through 56, and then to stop without processing any additional transactions.
`SQL_AFTER_MTS_GAPS`	For a multithreaded replica only (with `replica_parallel_workers` or `slave_parallel_workers` > 0), this option makes the replica process transactions up to the point where there are no more gaps in the sequence of transactions executed from the relay log. When using a multithreaded replica, there is a chance of gaps occurring in the following situations: • The coordinator thread is stopped. • An error occurs in the applier threads. • `mysqld` shuts down unexpectedly. When a replication channel has gaps, the replica's database is in a state that might never have existed on the source. The replica tracks the gaps internally and disallows `CHANGE REPLICATION`

`SOURCE TO` statements that would remove the gap information if they executed.

Before MySQL 8.0.26, issuing `START REPLICA` on a multithreaded replica with gaps in the sequence of transactions executed from the relay log generates a warning. To correct this situation, the solution is to use `START REPLICA UNTIL SQL_AFTER_MTS_GAPS`. See Section 17.5.1.34, “Replication and Transaction Inconsistencies” for more information.

From MySQL 8.0.26, the process of checking for gaps in the sequence of transactions is skipped entirely when GTID-based replication and GTID auto-positioning (`SOURCE_AUTO_POSITION=1`) are in use for the channel, because gaps in transactions can be resolved using GTID auto-positioning. In that situation, `START REPLICA UNTIL SQL_AFTER_MTS_GAPS` just stops the applier thread when it finds the first transaction to execute, and does not attempt to check for gaps in the sequence of transactions. You can also continue to use `CHANGE REPLICATION SOURCE TO` statements as normal, and relay log recovery is possible for the channel.

From MySQL 8.0.27, all replicas are multithreaded by default. When `replica_preserve_commit_order=ON` or `slave_preserve_commit_order=ON` is set for the replica, which is also the default setting from MySQL 8.0.27, gaps should not occur except in the specific situations listed in the description for `replica_preserve_commit_order` and `slave_preserve_commit_order`. If `replica_preserve_commit_order=OFF` or `slave_preserve_commit_order=OFF` is set for the replica, which is the default before MySQL 8.0.27, the commit order of transactions is not preserved, so the chance of gaps occurring is much larger.

If GTIDs are not in use and you need to change a failed multithreaded replica to single-threaded mode, you can issue the following series of statements, in the order shown:

```
START SLAVE UNTIL SQL_AFTER_MTS_GAPS;
SET @@GLOBAL.slave_parallel_workers = 0;
START SLAVE SQL_THREAD;

Or from MySQL 8.0.26:
START REPLICA UNTIL SQL_AFTER_MTS_GAPS;
SET @@GLOBAL.replica_parallel_workers = 0;
START REPLICA SQL_THREAD;
```

13.4.2.9 START SLAVE Statement

```
START {SLAVE | REPLICA} [thread_types] [until_option] [connection_options] [channel_option]

thread_types:
    [thread_type [, thread_type] ... ]

thread_type:
    IO_THREAD | SQL_THREAD

until_option:
    UNTIL {   {SQL_BEFORE_GTIDS | SQL_AFTER_GTIDS} = gtid_set
          |   MASTER_LOG_FILE = 'log_name', MASTER_LOG_POS = log_pos
          |   SOURCE_LOG_FILE = 'log_name', SOURCE_LOG_POS = log_pos
          |   RELAY_LOG_FILE = 'log_name', RELAY_LOG_POS = log_pos
```

```
          |   SQL_AFTER_MTS_GAPS  }

connection_options:
    [USER='user_name'] [PASSWORD='user_pass'] [DEFAULT_AUTH='plugin_name'] [PLUGIN_DIR='plugin_dir']


channel_option:
    FOR CHANNEL channel

gtid_set:
    uuid_set [, uuid_set] ...
    | ''

uuid_set:
    uuid:interval[:interval]...

uuid:
    hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh

h:
    [0-9,A-F]

interval:
    n[-n]

    (n >= 1)
```

Starts the replication threads. From MySQL 8.0.22, `START SLAVE` is deprecated and the alias `START REPLICA` should be used instead. The statement works in the same way as before, only the terminology used for the statement and its output has changed. Both versions of the statement update the same status variables when used. Please see the documentation for `START REPLICA` for a description of the statement.

13.4.2.10 STOP REPLICA Statement

```
STOP REPLICA [thread_types] [channel_option]

thread_types:
    [thread_type [, thread_type] ... ]

thread_type: IO_THREAD | SQL_THREAD

channel_option:
    FOR CHANNEL channel
```

Stops the replication threads. From MySQL 8.0.22, use `STOP REPLICA` in place of `STOP SLAVE`, which is now deprecated. In releases before MySQL 8.0.22, use `STOP SLAVE`.

`STOP REPLICA` requires the `REPLICATION_SLAVE_ADMIN` privilege (or the deprecated `SUPER` privilege). Recommended best practice is to execute `STOP REPLICA` on the replica before stopping the replica server (see Section 5.1.19, “The Server Shutdown Process”, for more information).

Like `START REPLICA`, this statement may be used with the `IO_THREAD` and `SQL_THREAD` options to name the replication thread or threads to be stopped. Note that the Group Replication applier channel (`group_replication_applier`) has no replication I/O (receiver) thread, only a replication SQL (applier) thread. Using the `SQL_THREAD` option therefore stops this channel completely.

`STOP REPLICA` causes an implicit commit of an ongoing transaction. See Section 13.3.3, “Statements That Cause an Implicit Commit”.

`gtid_next` must be set to `AUTOMATIC` before issuing this statement.

You can control how long `STOP REPLICA` waits before timing out by setting the system variable `rpl_stop_replica_timeout` (from MySQL 8.0.26) or `rpl_stop_slave_timeout` (before MySQL 8.0.26). This can be used to avoid deadlocks between `STOP REPLICA` and other SQL statements using different client connections to the replica. When the timeout value is reached, the issuing client

returns an error message and stops waiting, but the `STOP REPLICA` instruction remains in effect. Once the replication threads are no longer busy, the `STOP REPLICA` statement is executed and the replica stops.

Some `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statements are allowed while the replica is running, depending on the states of the replication threads. However, using `STOP REPLICA` prior to executing a `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement in such cases is still supported. See Section 13.4.2.3, “CHANGE REPLICATION SOURCE TO Statement”, Section 13.4.2.1, “CHANGE MASTER TO Statement”, and Section 17.4.8, “Switching Sources During Failover”, for more information.

The optional `FOR CHANNEL` *`channel`* clause enables you to name which replication channel the statement applies to. Providing a `FOR CHANNEL` *`channel`* clause applies the `STOP REPLICA` statement to a specific replication channel. If no channel is named and no extra channels exist, the statement applies to the default channel. If a `STOP REPLICA` statement does not name a channel when using multiple channels, this statement stops the specified threads for all channels. See Section 17.2.2, “Replication Channels” for more information.

The replication channels for Group Replication (`group_replication_applier` and `group_replication_recovery`) are managed automatically by the server instance. `STOP REPLICA` cannot be used at all with the `group_replication_recovery` channel, and should only be used with the `group_replication_applier` channel when Group Replication is not running. The `group_replication_applier` channel only has an applier thread and has no receiver thread, so it can be stopped if required by using the `SQL_THREAD` option without the `IO_THREAD` option.

When the replica is multithreaded (`replica_parallel_workers` or `slave_parallel_workers` is a nonzero value), any gaps in the sequence of transactions executed from the relay log are closed as part of stopping the worker threads. If the replica is stopped unexpectedly (for example due to an error in a worker thread, or another thread issuing `KILL`) while a `STOP REPLICA` statement is executing, the sequence of executed transactions from the relay log may become inconsistent. See Section 17.5.1.34, “Replication and Transaction Inconsistencies”, for more information.

When the source is using the row-based binary logging format, you should execute `STOP REPLICA` or `STOP REPLICA SQL_THREAD` on the replica prior to shutting down the replica server if you are replicating any tables that use a nontransactional storage engine. If the current replication event group has modified one or more nontransactional tables, `STOP REPLICA` waits for up to 60 seconds for the event group to complete, unless you issue a `KILL QUERY` or `KILL CONNECTION` statement for the replication SQL thread. If the event group remains incomplete after the timeout, an error message is logged.

When the source is using the statement-based binary logging format, changing the source while it has open temporary tables is potentially unsafe. This is one of the reasons why statement-based replication of temporary tables is not recommended. You can find out whether there are any temporary tables on the replica by checking the value of `Replica_open_temp_tables` or `Slave_open_temp_tables`. When using statement-based replication, this value should be 0 before executing `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO`. If there are any temporary tables open on the replica, issuing a `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement after issuing a `STOP REPLICA` causes an `ER_WARN_OPEN_TEMP_TABLES_MUST_BE_ZERO` warning.

13.4.2.11 STOP SLAVE Statement

```
STOP {SLAVE | REPLICA} [thread_types] [channel_option]

thread_types:
    [thread_type [, thread_type] ... ]

thread_type: IO_THREAD | SQL_THREAD

channel_option:
    FOR CHANNEL channel
```

Stops the replication threads. From MySQL 8.0.22, `STOP SLAVE` is deprecated and the alias `STOP REPLICA` should be used instead. The statement works in the same way as before, only the terminology used for the statement and its output has changed. Both versions of the statement update the same status variables when used. Please see the documentation for `STOP REPLICA` for a description of the statement.

13.4.2.12 Functions which Configure the Source List

The following functions, which are available from MySQL 8.0.22 for standard source to replica replication and from MySQL 8.0.23 for Group Replication, enable you to add and remove replication source servers from the source list for a replication channel. From MySQL 8.0.27, you can also clear the source list for a server.

The asynchronous connection failover mechanism automatically establishes an asynchronous (source to replica) replication connection to a new source from the appropriate list after the existing connection from the replica to its source fails. From MySQL 8.0.23, the connection is also changed if the currently connected source does not have the highest weighted priority in the group. For Group Replication source servers that are defined as part of a managed group, the connection is also failed over to another group member if the currently connected source leaves the group or is no longer in the majority. For more information on the mechanism, see Section 17.4.9, “Switching Sources and Replicas with Asynchronous Connection Failover”.

Source lists are stored in the `mysql.replication_asynchronous_connection_failover` and `mysql.replication_asynchronous_connection_failover_managed` tables, and can be viewed in the Performance Schema table `replication_asynchronous_connection_failover`.

If the replication channel is on a Group Replication primary for a group where failover between replicas is active, the source list is broadcast to all the group members when they join or when it is updated by any method. Failover between replicas is controlled by the `mysql_start_failover_channels_if_primary` member action, which is enabled by default, and can be disabled using the `group_replication_disable_member_action` function.

- `asynchronous_connection_failover_add_source()`

  Add configuration information for a replication source server to the source list for a replication channel.

  Syntax:

  ```
  asynchronous_connection_failover_add_source(channel, host, port, network_namespace, weight)
  ```

  Arguments:

  - *`channel`*: The replication channel for which this replication source server is part of the source list.
  - *`host`*: The host name for this replication source server.
  - *`port`*: The port number for this replication source server.
  - *`network_namespace`*: The network namespace for this replication source server. Specify an empty string, as this parameter is reserved for future use.
  - *`weight`*: The priority of this replication source server in the replication channel's source list. The priority is from 1 to 100, with 100 being the highest, and 50 being the default. When the asynchronous connection failover mechanism activates, the source with the highest priority setting among the alternative sources listed in the source list for the channel is chosen for the first connection attempt. If this attempt does not work, the replica tries with all the listed sources in descending order of priority, then starts again from the highest priority source. If multiple sources have the same priority, the replica orders them randomly. From MySQL 8.0.23, the asynchronous connection failover mechanism activates if the currently connected source is not the highest weighted in the group.

Return value:

A string containing the result of the operation, for example whether it was successful or not.

Example:

```
SELECT asynchronous_connection_failover_add_source('channel2', '127.0.0.1', 3310, '', 80);
+-----------------------------------------------------------------------------------------------+
| asynchronous_connection_failover_add_source('channel2', '127.0.0.1', 3310, '', 80)            |
+-----------------------------------------------------------------------------------------------+
| Source configuration details successfully inserted.                                           |
+-----------------------------------------------------------------------------------------------+
```

For more information, see Section 17.4.9, “Switching Sources and Replicas with Asynchronous Connection Failover”.

- `asynchronous_connection_failover_delete_source()`

  Remove configuration information for a replication source server from the source list for a replication channel.

  Syntax:

  ```
  asynchronous_connection_failover_delete_source(channel, host, port, network_namespace)
  ```

  Arguments:

  - *`channel`*: The replication channel for which this replication source server was part of the source list.
  - *`host`*: The host name for this replication source server.
  - *`port`*: The port number for this replication source server.
  - *`network_namespace`*: The network namespace for this replication source server. Specify an empty string, as this parameter is reserved for future use.

  Return value:

  A string containing the result of the operation, for example whether it was successful or not.

  Example:

  ```
  SELECT asynchronous_connection_failover_delete_source('channel2', '127.0.0.1', 3310, '');
  +-----------------------------------------------------------------------------------------------+
  | asynchronous_connection_failover_delete_source('channel2', '127.0.0.1', 3310, '')            |
  +-----------------------------------------------------------------------------------------------+
  | Source configuration details successfully deleted.                                            |
  +-----------------------------------------------------------------------------------------------+
  ```

  For more information, see Section 17.4.9, “Switching Sources and Replicas with Asynchronous Connection Failover”.

- `asynchronous_connection_failover_add_managed()`

  Add configuration information for a replication source server that is part of a managed group (a Group Replication group member) to the source list for a replication channel. You only need to add

one group member. The replica automatically adds the rest from the current group membership, then keeps the source list updated in line with membership change.

Syntax:

```
asynchronous_connection_failover_add_managed(channel, managed_type, managed_name, host, port, network_nam
```

Arguments:

- *`channel`*: The replication channel for which this replication source server is part of the source list.
- *`managed_type`*: The type of managed service that the asynchronous connection failover mechanism must provide for this server. The only value currently accepted is `GroupReplication`.
- *`managed_name`*: The identifier for the managed group that the server is a part of. For the `GroupReplication` managed service, the identifier is the value of the `group_replication_group_name` system variable.
- *`host`*: The host name for this replication source server.
- *`port`*: The port number for this replication source server.
- *`network_namespace`*: The network namespace for this replication source server. Specify an empty string, as this parameter is reserved for future use.
- *`primary_weight`*: The priority of this replication source server in the replication channel's source list when it is acting as the primary for the managed group. The weight is from 1 to 100, with 100 being the highest. For the primary, 80 is a suitable weight. The asynchronous connection failover mechanism activates if the currently connected source is not the highest weighted in the group. Assuming that you set up the managed group to give a higher weight to a primary and a lower weight to a secondary, when the primary changes, its weight increases, and the replica changes over the connection to it.
- *`secondary_weight`*: The priority of this replication source server in the replication channel's source list when it is acting as a secondary in the managed group. The weight is from 1 to 100, with 100 being the highest. For a secondary, 60 is a suitable weight.

Return value:

A string containing the result of the operation, for example whether it was successful or not.

Example:

```
SELECT asynchronous_connection_failover_add_managed('channel2', 'GroupReplication', 'aaaaaaaa-aaaa-aaaa-a
+-----------------------------------------------------------------------------------------------------------
| asynchronous_connection_failover_add_source('channel2', 'GroupReplication', 'aaaaaaaa-aaaa-aaaa-aaaa-aa
+-----------------------------------------------------------------------------------------------------------
| Source managed configuration details successfully inserted.
+-----------------------------------------------------------------------------------------------------------
```

For more information, see Section 17.4.9, “Switching Sources and Replicas with Asynchronous Connection Failover”.

- `asynchronous_connection_failover_delete_managed()`

  Remove an entire managed group from the source list for a replication channel. When you use this function, all the replication source servers defined in the managed group are removed from the channel's source list.

  Syntax:

  ```
  asynchronous_connection_failover_delete_managed(channel, managed_name)
  ```

  Arguments:

  - *`channel`*: The replication channel for which this replication source server was part of the source list.
  - *`managed_name`*: The identifier for the managed group that the server is a part of. For the `GroupReplication` managed service, the identifier is the value of the `group_replication_group_name` system variable.

  Return value:

  A string containing the result of the operation, for example whether it was successful or not.

  Example:

  ```
  SELECT asynchronous_connection_failover_delete_managed('channel2', 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaa
  +-----------------------------------------------------------------------------------------------------
  | asynchronous_connection_failover_delete_managed('channel2', 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa')
  +-----------------------------------------------------------------------------------------------------
  | Source managed configuration details successfully deleted.
  +-----------------------------------------------------------------------------------------------------
  ```

  For more information, see Section 17.4.9, “Switching Sources and Replicas with Asynchronous Connection Failover”.

- `asynchronous_connection_failover_reset()`

  Remove all settings relating to the asynchronous connection failover mechanism. The function clears the Performance Schema tables `replication_asynchronous_connection_failover` and `replication_asynchronous_connection_failover_managed`.

  `asynchronous_connection_failover_reset()` can be used only on a server that is not currently part of a group, and that does not have any replication channels running. You can use this function to clean up a server that is no longer being used in a managed group.

  Syntax:

  ```
  STRING asynchronous_connection_failover_reset()
  ```

  Arguments:

  None.

  Return value:

  A string containing the result of the operation, for example whether it was successful or not.

  Example:

  ```
  mysql> SELECT asynchronous_connection_failover_reset();
  +-----------------------------------------------------------------------+
  | asynchronous_connection_failover_reset()                              |
  +-----------------------------------------------------------------------+
  | The UDF asynchronous_connection_failover_reset() executed successfully. |
  ```

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

For more information, see Section 17.4.9, “Switching Sources and Replicas with Asynchronous Connection Failover”.

13.4.3 SQL Statements for Controlling Group Replication

This section provides information about the statements used for controlling group replication.

13.4.3.1 START GROUP_REPLICATION Statement

```
START GROUP_REPLICATION
        [USER='user_name']
        [, PASSWORD='user_pass']
        [, DEFAULT_AUTH='plugin_name']
```

Starts group replication. This statement requires the `GROUP_REPLICATION_ADMIN` privilege (or the deprecated `SUPER` privilege). If `super_read_only=ON` is set and the member should join as a primary, `super_read_only` is set to `OFF` once Group Replication successfully starts.

From MySQL 8.0.21, you can specify user credentials for distributed recovery on the `START GROUP_REPLICATION` statement using the `USER`, `PASSWORD`, and `DEFAULT_AUTH` options, as follows:

- `USER`: The replication user for distributed recovery. For instructions to set up this account, see Section 18.2.1.3, “User Credentials For Distributed Recovery”. You cannot specify an empty or null string, or omit the `USER` option if `PASSWORD` is specified.
- `PASSWORD`: The password for the replication user account. The password cannot be encrypted, but it is masked in the query log.
- `DEFAULT_AUTH`: The name of the authentication plugin used for the replication user account. If you do not specify this option, MySQL native authentication (the `mysql_native_password` plugin) is assumed. This option acts as a hint to the server, and the donor for distributed recovery overrides it if a different plugin is associated with the user account on that server. The authentication plugin used by default when you create user accounts in MySQL 8 is the caching SHA-2 authentication plugin (`caching_sha2_password`). See Section 6.2.17, “Pluggable Authentication” for more information on authentication plugins.

These credentials are used for distributed recovery on the `group_replication_recovery` channel. When you specify user credentials on `START GROUP_REPLICATION`, the credentials are saved in memory only, and are removed by a `STOP GROUP_REPLICATION` statement or server shutdown. You must issue a `START GROUP_REPLICATION` statement to provide the credentials again. This method is therefore not compatible with starting Group Replication automatically on server start, as specified by the `group_replication_start_on_boot` system variable.

User credentials specified on `START GROUP_REPLICATION` take precedence over any user credentials set for the `group_replication_recovery` channel using a `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23). Note that user credentials set using these statements are stored in the replication metadata repositories, and are used when `START GROUP_REPLICATION` is specified without user credentials, including automatic starts if the `group_replication_start_on_boot` system variable is set to `ON`. To gain the security benefits of specifying user credentials on `START GROUP_REPLICATION`, ensure that `group_replication_start_on_boot` is set to `OFF` (the default is `ON`), and clear any user credentials previously set for the `group_replication_recovery` channel, following the instructions in Section 18.6.3, “Securing Distributed Recovery Connections”.

While a member is rejoining a replication group, its status can be displayed as `OFFLINE` or `ERROR` before the group completes the compatibility checks and accepts it as a member. When the member is catching up with the group's transactions, its status is `RECOVERING`.

13.4.3.2 STOP GROUP_REPLICATION Statement

```
STOP GROUP_REPLICATION
```

Stops Group Replication. This statement requires the `GROUP_REPLICATION_ADMIN` privilege (or the deprecated `SUPER` privilege). As soon as you issue `STOP GROUP_REPLICATION` the member is set to `super_read_only=ON`, which ensures that no writes can be made to the member while Group Replication stops. Any other asynchronous replication channels running on the member are also stopped. Any user credentials that you specified in the `START GROUP_REPLICATION` statement when starting Group Replication on this member are removed from memory, and must be supplied when you start Group Replication again.

Warning

Use this statement with extreme caution because it removes the server instance from the group, meaning it is no longer protected by Group Replication's consistency guarantee mechanisms. To be completely safe, ensure that your applications can no longer connect to the instance before issuing this statement to avoid any chance of stale reads.

The `STOP GROUP_REPLICATION` statement stops asynchronous replication channels on the group member, but it does not implicitly commit transactions that are in progress on them like `STOP REPLICA` does. This is because on a Group Replication group member, an additional transaction committed during the shutdown operation would leave the member inconsistent with the group and cause an issue with rejoining. To avoid failed commits for transactions that are in progress while stopping Group Replication, from MySQL 8.0.28, the `STOP GROUP_REPLICATION` statement cannot be issued while a GTID is assigned as the value of the `gtid_next` system variable.

The `group_replication_components_stop_timeout` system variable specifies the time for which Group Replication waits for each of its modules to complete ongoing processes after this statement is issued. The timeout is used to resolve situations in which Group Replication components cannot be stopped normally, which can happen if the member is expelled from the group while it is in an error state, or while a process such as MySQL Enterprise Backup is holding a global lock on tables on the member. In such situations, the member cannot stop the applier thread or complete the distributed recovery process to rejoin. `STOP GROUP_REPLICATION` does not complete until either the situation is resolved (for example, by the lock being released), or the component timeout expires and the modules are shut down regardless of their status. Prior to MySQL 8.0.27, the default component timeout is 31536000 seconds, or 365 days. With this setting, the component timeout does not help in situations such as those just described, so a lower setting is recommended in those versions of MySQL 8.0. Beginning with MySQL 8.0.27, the default value is 300 seconds; this means that Group Replication components are stopped after 5 minutes if the situation is not resolved before that time, allowing the member to be restarted and rejoin.

13.4.3.3 Function which Configures Group Replication Primary

The following function enables you to set a member of a single-primary replication group to take over as the primary. The current primary becomes a read-only secondary, and the specified group member becomes the read-write primary. The function can be used on any member of a replication group running in single-primary mode. This function replaces the usual primary election process. Section 18.5.1.1, “Changing a Group's Primary Member” explains the MySQL Server version requirements for the new primary member.

If a standard source to replica replication channel is running on the existing primary member in addition to the Group Replication channels, you must stop that replication channel before you can change the primary member. You can identify the current primary using the `MEMBER_ROLE` column in the Performance Schema table `replication_group_members`, or the `group_replication_primary_member` status variable.

Any uncommitted transactions that the group is waiting on must be committed, rolled back, or terminated before the operation can complete. Before MySQL 8.0.29, the function waits for all active

transactions on the existing primary to end, including incoming transactions that are started after the function is used. From MySQL 8.0.29, you can specify a timeout for transactions that are running when you use the function. For the timeout to work, all members of the group must be at MySQL 8.0.29 or higher.

When the timeout expires, for any transactions that did not yet reach their commit phase, the client session is disconnected so that the transaction does not proceed. Transactions that reached their commit phase are allowed to complete. When you set a timeout, it also prevents new transactions starting on the primary from that point on. Explicitly defined transactions (with a `START TRANSACTION` or `BEGIN` statement) are subject to the timeout, disconnection, and incoming transaction blocking even if they do not modify any data. To allow inspection of the primary while the function is operating, single statements that do not modify data, as listed in Permitted Queries Under Consistency Rules, are permitted to proceed.

- `group_replication_set_as_primary()`

  Appoints a specific member of the group as the new primary, overriding any election process.

  Syntax:

  ```
  STRING group_replication_set_as_primary(member_uuid[, timeout])
  ```

  Arguments:

  - *`member_uuid`*: A string containing the UUID of the member of the group that you want to become the new primary.
  - *`timeout`*: An integer specifying a timeout in seconds for transactions that are running on the existing primary when you use the function. You can set a timeout from 0 seconds (immediately) up to 3600 seconds (60 minutes). When you set a timeout, new transactions cannot start on the primary from that point on. There is no default setting for the timeout, so if you do not set it, there is no upper limit to the wait time, and new transactions can start during that time. This option is available from MySQL 8.0.29.

  Return value:

  A string containing the result of the operation, for example whether it was successful or not.

  Example:

  ```
  SELECT group_replication_set_as_primary('00371d66-3c45-11ea-804b-080027337932', 300)
  ```

  For more information, see Section 18.5.1.1, "Changing a Group's Primary Member".

13.4.3.4 Functions which Configure the Group Replication Mode

The following functions enable you to control the mode which a replication group is running in, either single-primary or multi-primary mode.

- `group_replication_switch_to_single_primary_mode()`

  Changes a group running in multi-primary mode to single-primary mode, without the need to stop Group Replication. Must be issued on a member of a replication group running in multi-primary mode. When you change to single-primary mode, strict consistency checks are also disabled on all group members, as required in single-primary mode (`group_replication_enforce_update_everywhere_checks=OFF`).

  Syntax:

  ```
  STRING group_replication_switch_to_single_primary_mode([str])
  ```

  Arguments:

- *`str`*: A string containing the UUID of a member of the group which should become the new single primary. Other members of the group become secondaries.

Return value:

A string containing the result of the operation, for example whether it was successful or not.

Example:

```
SELECT group_replication_switch_to_single_primary_mode(member_uuid);
```

For more information, see Section 18.5.1.2, “Changing a Group's Mode”

- `group_replication_switch_to_multi_primary_mode()`

Changes a group running in single-primary mode to multi-primary mode. Must be issued on a member of a replication group running in single-primary mode.

Syntax:

```
STRING group_replication_switch_to_multi_primary_mode()
```

This function has no parameters.

Return value:

A string containing the result of the operation, for example whether it was successful or not.

Example:

```
SELECT group_replication_switch_to_multi_primary_mode()
```

All members which belong to the group become primaries.

For more information, see Section 18.5.1.2, “Changing a Group's Mode”

13.4.3.5 Functions to Inspect and Configure the Maximum Consensus Instances of a Group

The following functions enable you to inspect and configure the maximum number of consensus instances that a group can execute in parallel.

- `group_replication_get_write_concurrency()`

Check the maximum number of consensus instances that a group can execute in parallel.

Syntax:

```
INT group_replication_get_write_concurrency()
```

This function has no parameters.

Return value:

The maximum number of consensus instances currently set for the group.

Example:

```
SELECT group_replication_get_write_concurrency()
```

For more information, see Section 18.5.1.3, “Using Group Replication Group Write Consensus”.

- `group_replication_set_write_concurrency()`

Configures the maximum number of consensus instances that a group can execute in parallel. The `GROUP_REPLICATION_ADMIN` privilege is required to use this function.

Syntax:

```
STRING group_replication_set_write_concurrency(instances)
```

Arguments:

- *`members`*: Sets the maximum number of consensus instances that a group can execute in parallel. Default value is 10, valid values are integers in the range of 10 to 200.

Return value:

Any resulting error as a string.

Example:

```
SELECT group_replication_set_write_concurrency(instances);
```

For more information, see Section 18.5.1.3, “Using Group Replication Group Write Consensus”.

13.4.3.6 Functions to Inspect and Set the Group Replication Communication Protocol Version

The following functions enable you to inspect and configure the Group Replication communication protocol version that is used by a replication group.

- Versions from MySQL 5.7.14 allow compression of messages (see Section 18.7.4, “Message Compression”).
- Versions from MySQL 8.0.16 also allow fragmentation of messages (see Section 18.7.5, “Message Fragmentation”).
- Versions from MySQL 8.0.27 also allow the group communication engine to operate with a single consensus leader when the group is in single-primary mode and `group_replication_paxos_single_leader` is set to true (see Section 18.7.3, “Single Consensus Leader”).
- `group_replication_get_communication_protocol()`

  Inspect the Group Replication communication protocol version that is currently in use for a group.

  Syntax:

  ```
  STRING group_replication_get_communication_protocol()
  ```

  This function has no parameters.

  Return value:

  The oldest MySQL Server version that can join this group and use the group's communication protocol. Note that the `group_replication_get_communication_protocol()` function returns the minimum MySQL version that the group supports, which might differ from the version number that was passed to the `group_replication_set_communication_protocol()` function, and from the MySQL Server version that is installed on the member where you use the function.

  If the protocol cannot be inspected because this server instance does not belong to a replication group, an error is returned as a string.

  Example:

```
SELECT group_replication_get_communication_protocol();
+------------------------------------------------+
| group_replication_get_communication_protocol() |
+------------------------------------------------+
| 8.0.16                                         |
+------------------------------------------------+
```

For more information, see Section 18.5.1.4, “Setting a Group's Communication Protocol Version”.

- `group_replication_set_communication_protocol()`

  Downgrade the Group Replication communication protocol version of a group so that members at earlier releases can join, or upgrade the Group Replication communication protocol version of a group after upgrading MySQL Server on all members. The `GROUP_REPLICATION_ADMIN` privilege is required to use this function, and all existing group members must be online when you issue the statement, with no loss of majority.

  

  Note

  For MySQL InnoDB cluster, the communication protocol version is managed automatically whenever the cluster topology is changed using AdminAPI operations. You do not have to use these functions yourself for an InnoDB cluster.

  Syntax:

  ```
  STRING group_replication_set_communication_protocol(version)
  ```

  Arguments:

  - *`version`*: For a downgrade, specify the MySQL Server version of the prospective group member that has the oldest installed server version. In this case, the command makes the group fall back to a communication protocol compatible with that server version if possible. The minimum server version that you can specify is MySQL 5.7.14. For an upgrade, specify the new MySQL Server version to which the existing group members have been upgraded.

  Return value:

  A string containing the result of the operation, for example whether it was successful or not.

  Example:

  ```
  SELECT group_replication_set_communication_protocol("5.7.25");
  ```

  For more information, see Section 18.5.1.4, “Setting a Group's Communication Protocol Version”.

13.4.3.7 Functions to Set and Reset Group Replication Member Actions

The following functions can be used to enable and disable actions for members of a group to take in specified situations, and to reset the configuration to the default setting for all member actions. They can only be used by administrators with the `GROUP_REPLICATION_ADMIN` privilege or the deprecated `SUPER` privilege.

You configure member actions on the group’s primary using the `group_replication_enable_member_action` and `group_replication_disable_member_action` functions. The member actions configuration, consisting of all the member actions and whether they are enabled or disabled, is then propagated to other group members and joining members using Group Replication’s group messages. This means that the group members will all act in the same way when they are in the specified situation, and you only need to use the function on the primary.

The functions can also be used on a server that is not part of a group, as long as the Group Replication plugin is installed. In that case, the member actions configuration is not propagated to any other servers.

The `group_replication_reset_member_actions` function can only be used on a server that is not part of a group. It resets the member actions configuration to the default settings, and resets its version number. The server must be writeable (with the `read_only` system variable set to `OFF`) and have the Group Replication plugin installed.

The available member actions are as follows:

`mysql_disable_super_read_only_if_primary`: This member action is available from MySQL 8.0.26. It is taken after a member is elected as the group's primary, which is the event `AFTER_PRIMARY_ELECTION`. The member action is enabled by default. You can disable it using the `group_replication_disable_member_action()` function, and re-enable it using the `group_replication_enable_member_action()` function.

When this member action is enabled and taken, super read-only mode is disabled on the primary, so that the primary becomes read-write and accepts updates from a replication source server and from clients. This is the normal situation.

When this member action is disabled and not taken, the primary remains in super read-only mode after election. In this state, it does not accept updates from any clients, even users who have the `CONNECTION_ADMIN` or `SUPER` privilege. It does continue to accept updates performed by replication threads. This setup means that when a group's purpose is to provide a secondary backup to another group for disaster tolerance, you can ensure that the secondary group remains synchronized with the first.

`mysql_start_failover_channels_if_primary`: This member action is available from MySQL 8.0.27. It is taken after a member is elected as the group's primary, which is the event `AFTER_PRIMARY_ELECTION`. The member action is enabled by default. You can disable it using the `group_replication_disable_member_action()` function, and re-enable it using the `group_replication_enable_member_action()` function.

When this member action is enabled, asynchronous connection failover for replicas is active for a replication channel on a Group Replication primary when you set `SOURCE_CONNECTION_AUTO_FAILOVER=1` on the `CHANGE REPLICATION SOURCE TO` statement for the channel. When the feature is active and correctly configured, if the primary that is replicating goes offline or into an error state, the new primary starts replication on the same channel when it is elected. This is the normal situation. For instructions to configure the feature, see Section 17.4.9.2, "Asynchronous Connection Failover for Replicas".

When this member action is disabled, asynchronous connection failover does not take place for the replicas. If the primary goes offline or into an error state, replication stops for the channel. Note that if there is more than one channel with `SOURCE_CONNECTION_AUTO_FAILOVER=1`, the member action covers all the channels, so they cannot be individually enabled and disabled by this method. Set

`SOURCE_CONNECTION_AUTO_FAILOVER=0` to disable an individual channel.

For more information on member actions and how to view the member actions configuration, see Section 18.5.1.5, “Configuring Member Actions”.

- `group_replication_enable_member_action()`

  Enable a member action for the member to take in the specified situation. If the server where you use the function is part of a group, it must be the current primary in a group in single-primary mode, and it must be part of the majority. The changed setting is propagated to other group members and joining members, so they will all act in the same way when they are in the specified situation, and you only need to use the function on the primary.

  Syntax:

  ```
  STRING group_replication_enable_member_action(name, event)
  ```

  Arguments:

  - *`name`*: The name of the member action to enable.
  - *`event`*: The event that triggers the member action.

  Return value:

  A string containing the result of the operation, for example whether it was successful or not.

  Example:

  ```
  SELECT group_replication_enable_member_action("mysql_disable_super_read_only_if_primary", "AFTER_PRIM
  ```

  For more information, see Section 18.5.1.5, “Configuring Member Actions”.

- `group_replication_disable_member_action()`

  Disable a member action so that the member does not take it in the specified situation. If the server where you use the function is part of a group, it must be the current primary in a group in single-primary mode, and it must be part of the majority. The changed setting is propagated to other group members and joining members, so they will all act in the same way when they are in the specified situation, and you only need to use the function on the primary.

  Syntax:

  ```
  STRING group_replication_disable_member_action(name, event)
  ```

  Arguments:

  - *`name`*: The name of the member action to disable.
  - *`event`*: The event that triggers the member action.

  Return value:

  A string containing the result of the operation, for example whether it was successful or not.

  Example:

  ```
  SELECT group_replication_disable_member_action("mysql_disable_super_read_only_if_primary", "AFTER_PRI
  ```

  For more information, see Section 18.5.1.5, “Configuring Member Actions”.

- `group_replication_reset_member_actions()`

Reset the member actions configuration to the default settings, and reset its version number to 1.

The `group_replication_reset_member_actions()` function can only be used on a server that is not currently part of a group. The server must be writeable (with the `read_only` system variable set to `OFF`) and have the Group Replication plugin installed. You can use this function to remove the member actions configuration that a server used when it was part of a group, if you intend to use it as a standalone server with no member actions or different member actions.

Syntax:

```
STRING group_replication_reset_member_actions()
```

Arguments:

None.

Return value:

A string containing the result of the operation, for example whether it was successful or not.

Example:

```
SELECT group_replication_reset_member_actions();
```

For more information, see Section 18.5.1.5, “Configuring Member Actions”.

13.5 Prepared Statements

MySQL 8.0 provides support for server-side prepared statements. This support takes advantage of the efficient client/server binary protocol. Using prepared statements with placeholders for parameter values has the following benefits:

- Less overhead for parsing the statement each time it is executed. Typically, database applications process large volumes of almost-identical statements, with only changes to literal or variable values in clauses such as `WHERE` for queries and deletes, `SET` for updates, and `VALUES` for inserts.
- Protection against SQL injection attacks. The parameter values can contain unescaped SQL quote and delimiter characters.

The following sections provide an overview of the characteristics of prepared statements:

- Prepared Statements in Application Programs
- Prepared Statements in SQL Scripts
- PREPARE, EXECUTE, and DEALLOCATE PREPARE Statements
- SQL Syntax Permitted in Prepared Statements

Prepared Statements in Application Programs

You can use server-side prepared statements through client programming interfaces, including the MySQL C API client library for C programs, MySQL Connector/J for Java programs, and MySQL Connector/NET for programs using .NET technologies. For example, the C API provides a set of function calls that make up its prepared statement API. See C API Prepared Statement Interface. Other language interfaces can provide support for prepared statements that use the binary protocol by linking in the C client library, one example being the `mysqli` extension, available in PHP 5.0 and higher.

Prepared Statements in SQL Scripts

An alternative SQL interface to prepared statements is available. This interface is not as efficient as using the binary protocol through a prepared statement API, but requires no programming because it is available directly at the SQL level:

- You can use it when no programming interface is available to you.
- You can use it from any program that can send SQL statements to the server to be executed, such as the `mysql` client program.
- You can use it even if the client is using an old version of the client library.

SQL syntax for prepared statements is intended to be used for situations such as these:

- To test how prepared statements work in your application before coding it.
- To use prepared statements when you do not have access to a programming API that supports them.
- To interactively troubleshoot application issues with prepared statements.
- To create a test case that reproduces a problem with prepared statements, so that you can file a bug report.

PREPARE, EXECUTE, and DEALLOCATE PREPARE Statements

SQL syntax for prepared statements is based on three SQL statements:

- `PREPARE` prepares a statement for execution (see Section 13.5.1, “PREPARE Statement”).
- `EXECUTE` executes a prepared statement (see Section 13.5.2, “EXECUTE Statement”).
- `DEALLOCATE PREPARE` releases a prepared statement (see Section 13.5.3, “DEALLOCATE PREPARE Statement”).

The following examples show two equivalent ways of preparing a statement that computes the hypotenuse of a triangle given the lengths of the two sides.

The first example shows how to create a prepared statement by using a string literal to supply the text of the statement:

```
mysql> PREPARE stmt1 FROM 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
mysql> SET @a = 3;
mysql> SET @b = 4;
mysql> EXECUTE stmt1 USING @a, @b;
+------------+
| hypotenuse |
+------------+
|          5 |
+------------+
mysql> DEALLOCATE PREPARE stmt1;
```

The second example is similar, but supplies the text of the statement as a user variable:

```
mysql> SET @s = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
mysql> PREPARE stmt2 FROM @s;
mysql> SET @a = 6;
mysql> SET @b = 8;
mysql> EXECUTE stmt2 USING @a, @b;
+------------+
| hypotenuse |
+------------+
|         10 |
+------------+
mysql> DEALLOCATE PREPARE stmt2;
```

Here is an additional example that demonstrates how to choose the table on which to perform a query at runtime, by storing the name of the table as a user variable:

```
mysql> USE test;
mysql> CREATE TABLE t1 (a INT NOT NULL);
mysql> INSERT INTO t1 VALUES (4), (8), (11), (32), (80);

mysql> SET @table = 't1';
mysql> SET @s = CONCAT('SELECT * FROM ', @table);

mysql> PREPARE stmt3 FROM @s;
mysql> EXECUTE stmt3;
+----+
| a  |
+----+
|  4 |
|  8 |
| 11 |
| 32 |
| 80 |
+----+

mysql> DEALLOCATE PREPARE stmt3;
```

A prepared statement is specific to the session in which it was created. If you terminate a session without deallocating a previously prepared statement, the server deallocates it automatically.

A prepared statement is also global to the session. If you create a prepared statement within a stored routine, it is not deallocated when the stored routine ends.

To guard against too many prepared statements being created simultaneously, set the `max_prepared_stmt_count` system variable. To prevent the use of prepared statements, set the value to 0.

SQL Syntax Permitted in Prepared Statements

The following SQL statements can be used as prepared statements:

```
ALTER TABLE
ALTER USER
ANALYZE TABLE
CACHE INDEX
CALL
CHANGE MASTER
CHECKSUM {TABLE | TABLES}
COMMIT
{CREATE | DROP} INDEX
{CREATE | RENAME | DROP} DATABASE
{CREATE | DROP} TABLE
{CREATE | RENAME | DROP} USER
{CREATE | DROP} VIEW
DELETE
DO
FLUSH {TABLE | TABLES | TABLES WITH READ LOCK | HOSTS | PRIVILEGES
  | LOGS | STATUS | MASTER | SLAVE | USER_RESOURCES}
GRANT
INSERT
INSTALL PLUGIN
KILL
LOAD INDEX INTO CACHE
OPTIMIZE TABLE
RENAME TABLE
REPAIR TABLE
REPLACE
RESET {MASTER | SLAVE}
REVOKE
SELECT
SET
SHOW BINLOG EVENTS
```

```
SHOW CREATE {PROCEDURE | FUNCTION | EVENT | TABLE | VIEW}
SHOW {MASTER | BINARY} LOGS
SHOW {MASTER | SLAVE} STATUS
SLAVE {START | STOP}
TRUNCATE TABLE
UNINSTALL PLUGIN
UPDATE
```

Other statements are not supported.

For compliance with the SQL standard, which states that diagnostics statements are not preparable, MySQL does not support the following as prepared statements:

- `SHOW WARNINGS`, `SHOW COUNT(*) WARNINGS`
- `SHOW ERRORS`, `SHOW COUNT(*) ERRORS`
- Statements containing any reference to the `warning_count` or `error_count` system variable.

Generally, statements not permitted in SQL prepared statements are also not permitted in stored programs. Exceptions are noted in Section 25.8, “Restrictions on Stored Programs”.

Metadata changes to tables or views referred to by prepared statements are detected and cause automatic repreparation of the statement when it is next executed. For more information, see Section 8.10.3, “Caching of Prepared Statements and Stored Programs”.

Placeholders can be used for the arguments of the `LIMIT` clause when using prepared statements. See Section 13.2.13, “SELECT Statement”.

In prepared `CALL` statements used with `PREPARE` and `EXECUTE`, placeholder support for `OUT` and `INOUT` parameters is available beginning with MySQL 8.0. See Section 13.2.1, “CALL Statement”, for an example and a workaround for earlier versions. Placeholders can be used for `IN` parameters regardless of version.

SQL syntax for prepared statements cannot be used in nested fashion. That is, a statement passed to `PREPARE` cannot itself be a `PREPARE`, `EXECUTE`, or `DEALLOCATE PREPARE` statement.

SQL syntax for prepared statements is distinct from using prepared statement API calls. For example, you cannot use the `mysql_stmt_prepare()` C API function to prepare a `PREPARE`, `EXECUTE`, or `DEALLOCATE PREPARE` statement.

SQL syntax for prepared statements can be used within stored procedures, but not in stored functions or triggers. However, a cursor cannot be used for a dynamic statement that is prepared and executed with `PREPARE` and `EXECUTE`. The statement for a cursor is checked at cursor creation time, so the statement cannot be dynamic.

SQL syntax for prepared statements does not support multi-statements (that is, multiple statements within a single string separated by `;` characters).

To write C programs that use the `CALL` SQL statement to execute stored procedures that contain prepared statements, the `CLIENT_MULTI_RESULTS` flag must be enabled. This is because each `CALL` returns a result to indicate the call status, in addition to any result sets that might be returned by statements executed within the procedure.

`CLIENT_MULTI_RESULTS` can be enabled when you call `mysql_real_connect()`, either explicitly by passing the `CLIENT_MULTI_RESULTS` flag itself, or implicitly by passing `CLIENT_MULTI_STATEMENTS` (which also enables `CLIENT_MULTI_RESULTS`). For additional information, see Section 13.2.1, “CALL Statement”.

13.5.1 PREPARE Statement

```
PREPARE stmt_name FROM preparable_stmt
```

The PREPARE statement prepares a SQL statement and assigns it a name, *stmt_name*, by which to refer to the statement later. The prepared statement is executed with EXECUTE and released with DEALLOCATE PREPARE. For examples, see Section 13.5, “Prepared Statements”.

Statement names are not case-sensitive. *preparable_stmt* is either a string literal or a user variable that contains the text of the SQL statement. The text must represent a single statement, not multiple statements. Within the statement, ? characters can be used as parameter markers to indicate where data values are to be bound to the query later when you execute it. The ? characters should not be enclosed within quotation marks, even if you intend to bind them to string values. Parameter markers can be used only where data values should appear, not for SQL keywords, identifiers, and so forth.

If a prepared statement with the given name already exists, it is deallocated implicitly before the new statement is prepared. This means that if the new statement contains an error and cannot be prepared, an error is returned and no statement with the given name exists.

The scope of a prepared statement is the session within which it is created, which as several implications:

- A prepared statement created in one session is not available to other sessions.
- When a session ends, whether normally or abnormally, its prepared statements no longer exist. If auto-reconnect is enabled, the client is not notified that the connection was lost. For this reason, clients may wish to disable auto-reconnect. See Automatic Reconnection Control.
- A prepared statement created within a stored program continues to exist after the program finishes executing and can be executed outside the program later.
- A statement prepared in stored program context cannot refer to stored procedure or function parameters or local variables because they go out of scope when the program ends and would be unavailable were the statement to be executed later outside the program. As a workaround, refer instead to user-defined variables, which also have session scope; see Section 9.4, “User-Defined Variables”.

Beginning with MySQL 8.0.22, a parameter used in a prepared statement has its type determined when the statement is first prepared, and retains this type whenever EXECUTE is invoked for this prepared statement (unless the statement is reprepared, as explained later in this section). Rules for determining a parameter's type are listed here:

- A parameter which is an operand of a binary arithmetic operator has the same data type as the other operand.
- If both operands of a binary arithmetic operator are parameters, the type of the parameters is decided by the context of the operator.
- If a parameter is the operand of a unary arithmetic operator, the parameter's type is decided by the context of the operator.
- If an arithmetic operator has no type-determining context, the derived type for any parameters involved is DOUBLE PRECISION. This can happen, for example, when the parameter is a top-level node in a SELECT list, or when it is part of a comparison operator.
- A parameter which is an operand of a character string operator has the same derived type as the aggregated type of the other operands. If all operands of the operator are parameters, the derived type is VARCHAR; its collation is determined by the value of collation_connection.
- A parameter which is an operand of a temporal operator has type DATETIME if the operator returns a DATETIME, TIME if the operator returns a TIME, and DATE if the operator returns a DATE.
- A parameter which is an operand of a binary comparison operator has the same derived type as the other operand of the comparison.
- A parameter that is an operand of a ternary comparison operator such as BETWEEN has the same derived type as the aggregated type of the other operands.

- If all operands of a comparison operator are parameters, the derived type for each of them is `VARCHAR`, with collation determined by the value of `collation_connection`.
- A parameter that is an output operand of any of `CASE`, `COALESCE`, `IF`, `IFNULL`, or `NULLIF` has the same derived type as the aggregated type of the operator's other output operands.
- If all output operands of any of `CASE`, `COALESCE`, `IF`, `IFNULL`, or `NULLIF` are parameters, or they are all `NULL`, the type of the parameter is decided by the context of the operator.
- If the parameter is an operand of any of `CASE`, `COALESCE()`, `IF`, or `IFNULL`, and has no type-determining context, the derived type for each of the parameters involved is `VARCHAR`, and its collation is determined by the value of `collation_connection`.
- A parameter which is the operand of a `CAST()` has the same type as specified by the `CAST()`.
- If a parameter is an immediate member of a `SELECT` list that is not part of an `INSERT` statement, the derived type of the parameter is `VARCHAR`, and its collation is determined by the value of `collation_connection`.
- If a parameter is an immediate member of a `SELECT` list that is part of an `INSERT` statement, the derived type of the parameter is the type of the corresponding column into which the parameter is inserted.
- If a parameter is used as source for an assignment in a `SET` clause of an `UPDATE` statement or in the `ON DUPLICATE KEY UPDATE` clause of an `INSERT` statement, the derived type of the parameter is the type of the corresponding column which is updated by the `SET` or `ON DUPLICATE KEY UPDATE` clause.
- If a parameter is an argument of a function, the derived type depends on the function's return type.

For some combinations of actual type and derived type, an automatic repreparation of the statement is triggered, to ensure closer compatibility with previous versions of MySQL. Repreparation does not occur if any of the following conditions are true:

- `NULL` is used as the actual parameter value.
- A parameter is an operand of a `CAST()`. (Instead, a cast to the derived type is attempted, and an exception raised if the cast fails.)
- A parameter is a string. (In this case, an implicit `CAST(? AS derived_type)` is performed.)
- The derived type and actual type of the parameter are both `INTEGER` and have the same sign.
- The parameter's derived type is `DECIMAL` and its actual type is either `DECIMAL` or `INTEGER`.
- The derived type is `DOUBLE` and the actual type is any numeric type.
- Both the derived type and the actual type are string types.
- If the derived type is temporal and the actual type is temporal. *Exceptions*: The derived type is `TIME` and the actual type is not `TIME`; the derived type is `DATE` and the actual type is not `DATE`.
- The derived type is temporal and the actual type is numeric.

For cases other than those just listed, the statement is reprepared and the actual parameter types are used instead of the derived parameter types.

These rules also apply to a user variable referenced in a prepared statement.

Using a different data type for a given parameter or user variable within a prepared statement for executions of the statement subsequent to the first execution causes the statement to be reprepared. This is less efficient; it may also lead to the parameter's (or variable's) actual type to vary, and thus for results to be inconsistent, with subsequent executions of the prepared statement. For these reasons, it is advisable to use the same data type for a given parameter when re-executing a prepared statement.

13.5.2 EXECUTE Statement

```
EXECUTE stmt_name
    [USING @var_name [, @var_name] ...]
```

After preparing a statement with `PREPARE`, you execute it with an `EXECUTE` statement that refers to the prepared statement name. If the prepared statement contains any parameter markers, you must supply a `USING` clause that lists user variables containing the values to be bound to the parameters. Parameter values can be supplied only by user variables, and the `USING` clause must name exactly as many variables as the number of parameter markers in the statement.

You can execute a given prepared statement multiple times, passing different variables to it or setting the variables to different values before each execution.

For examples, see Section 13.5, “Prepared Statements”.

13.5.3 DEALLOCATE PREPARE Statement

```
{DEALLOCATE | DROP} PREPARE stmt_name
```

To deallocate a prepared statement produced with `PREPARE`, use a `DEALLOCATE PREPARE` statement that refers to the prepared statement name. Attempting to execute a prepared statement after deallocating it results in an error. If too many prepared statements are created and not deallocated by either the `DEALLOCATE PREPARE` statement or the end of the session, you might encounter the upper limit enforced by the `max_prepared_stmt_count` system variable.

For examples, see Section 13.5, “Prepared Statements”.

13.6 Compound Statement Syntax

This section describes the syntax for the `BEGIN ... END` compound statement and other statements that can be used in the body of stored programs: Stored procedures and functions, triggers, and events. These objects are defined in terms of SQL code that is stored on the server for later invocation (see Chapter 25, *Stored Objects*).

A compound statement is a block that can contain other blocks; declarations for variables, condition handlers, and cursors; and flow control constructs such as loops and conditional tests.

13.6.1 BEGIN ... END Compound Statement

```
[begin_label:] BEGIN
    [statement_list]
END [end_label]
```

`BEGIN ... END` syntax is used for writing compound statements, which can appear within stored programs (stored procedures and functions, triggers, and events). A compound statement can contain multiple statements, enclosed by the `BEGIN` and `END` keywords. *`statement_list`* represents a list of one or more statements, each terminated by a semicolon (`;`) statement delimiter. The *`statement_list`* itself is optional, so the empty compound statement (`BEGIN END`) is legal.

`BEGIN ... END` blocks can be nested.

Use of multiple statements requires that a client is able to send statement strings containing the `;` statement delimiter. In the `mysql` command-line client, this is handled with the `delimiter` command. Changing the `;` end-of-statement delimiter (for example, to `//`) permit `;` to be used in a program body. For an example, see Section 25.1, “Defining Stored Programs”.

A `BEGIN ... END` block can be labeled. See Section 13.6.2, “Statement Labels”.

The optional `[NOT] ATOMIC` clause is not supported. This means that no transactional savepoint is set at the start of the instruction block and the `BEGIN` clause used in this context has no effect on the current transaction.

Note

Within all stored programs, the parser treats `BEGIN [WORK]` as the beginning of a `BEGIN ... END` block. To begin a transaction in this context, use `START TRANSACTION` instead.

13.6.2 Statement Labels

```
[begin_label:] BEGIN
    [statement_list]
END [end_label]

[begin_label:] LOOP
    statement_list
END LOOP [end_label]

[begin_label:] REPEAT
    statement_list
UNTIL search_condition
END REPEAT [end_label]

[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```

Labels are permitted for `BEGIN ... END` blocks and for the `LOOP`, `REPEAT`, and `WHILE` statements. Label use for those statements follows these rules:

- *`begin_label`* must be followed by a colon.
- *`begin_label`* can be given without *`end_label`*. If *`end_label`* is present, it must be the same as *`begin_label`*.
- *`end_label`* cannot be given without *`begin_label`*.
- Labels at the same nesting level must be distinct.
- Labels can be up to 16 characters long.

To refer to a label within the labeled construct, use an `ITERATE` or `LEAVE` statement. The following example uses those statements to continue iterating or terminate the loop:

```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
  label1: LOOP
    SET p1 = p1 + 1;
    IF p1 < 10 THEN ITERATE label1; END IF;
    LEAVE label1;
  END LOOP label1;
END;
```

The scope of a block label does not include the code for handlers declared within the block. For details, see Section 13.6.7.2, “DECLARE ... HANDLER Statement”.

13.6.3 DECLARE Statement

The `DECLARE` statement is used to define various items local to a program:

- Local variables. See Section 13.6.4, “Variables in Stored Programs”.
- Conditions and handlers. See Section 13.6.7, “Condition Handling”.
- Cursors. See Section 13.6.6, “Cursors”.

`DECLARE` is permitted only inside a `BEGIN ... END` compound statement and must be at its start, before any other statements.

Declarations must follow a certain order. Cursor declarations must appear before handler declarations. Variable and condition declarations must appear before cursor or handler declarations.

13.6.4 Variables in Stored Programs

System variables and user-defined variables can be used in stored programs, just as they can be used outside stored-program context. In addition, stored programs can use `DECLARE` to define local variables, and stored routines (procedures and functions) can be declared to take parameters that communicate values between the routine and its caller.

- To declare local variables, use the `DECLARE` statement, as described in Section 13.6.4.1, “Local Variable DECLARE Statement”.
- Variables can be set directly with the `SET` statement. See Section 13.7.6.1, “SET Syntax for Variable Assignment”.
- Results from queries can be retrieved into local variables using `SELECT ... INTO var_list` or by opening a cursor and using `FETCH ... INTO var_list`. See Section 13.2.13.1, “SELECT ... INTO Statement”, and Section 13.6.6, “Cursors”.

For information about the scope of local variables and how MySQL resolves ambiguous names, see Section 13.6.4.2, “Local Variable Scope and Resolution”.

It is not permitted to assign the value `DEFAULT` to stored procedure or function parameters or stored program local variables (for example with a `SET var_name = DEFAULT` statement). In MySQL 8.0, this results in a syntax error.

13.6.4.1 Local Variable DECLARE Statement

```
DECLARE var_name [, var_name] ... type [DEFAULT value]
```

This statement declares local variables within stored programs. To provide a default value for a variable, include a `DEFAULT` clause. The value can be specified as an expression; it need not be a constant. If the `DEFAULT` clause is missing, the initial value is `NULL`.

Local variables are treated like stored routine parameters with respect to data type and overflow checking. See Section 13.1.17, “CREATE PROCEDURE and CREATE FUNCTION Statements”.

Variable declarations must appear before cursor or handler declarations.

Local variable names are not case-sensitive. Permissible characters and quoting rules are the same as for other identifiers, as described in Section 9.2, “Schema Object Names”.

The scope of a local variable is the `BEGIN ... END` block within which it is declared. The variable can be referred to in blocks nested within the declaring block, except those blocks that declare a variable with the same name.

For examples of variable declarations, see Section 13.6.4.2, “Local Variable Scope and Resolution”.

13.6.4.2 Local Variable Scope and Resolution

The scope of a local variable is the `BEGIN ... END` block within which it is declared. The variable can be referred to in blocks nested within the declaring block, except those blocks that declare a variable with the same name.

Because local variables are in scope only during stored program execution, references to them are not permitted in prepared statements created within a stored program. Prepared statement scope is the current session, not the stored program, so the statement could be executed after the program ends, at which point the variables would no longer be in scope. For example, `SELECT ... INTO local_var` cannot be used as a prepared statement. This restriction also applies to stored procedure and function parameters. See Section 13.5.1, “PREPARE Statement”.

A local variable should not have the same name as a table column. If an SQL statement, such as a `SELECT ... INTO` statement, contains a reference to a column and a declared local variable with the same name, MySQL currently interprets the reference as the name of a variable. Consider the following procedure definition:

```
CREATE PROCEDURE sp1 (x VARCHAR(5))
BEGIN
  DECLARE xname VARCHAR(5) DEFAULT 'bob';
  DECLARE newname VARCHAR(5);
  DECLARE xid INT;

  SELECT xname, id INTO newname, xid
    FROM table1 WHERE xname = xname;
  SELECT newname;
END;
```

MySQL interprets `xname` in the `SELECT` statement as a reference to the `xname` *variable* rather than the `xname` *column*. Consequently, when the procedure `sp1()`is called, the `newname` variable returns the value `'bob'` regardless of the value of the `table1.xname` column.

Similarly, the cursor definition in the following procedure contains a `SELECT` statement that refers to `xname`. MySQL interprets this as a reference to the variable of that name rather than a column reference.

```
CREATE PROCEDURE sp2 (x VARCHAR(5))
BEGIN
  DECLARE xname VARCHAR(5) DEFAULT 'bob';
  DECLARE newname VARCHAR(5);
  DECLARE xid INT;
  DECLARE done TINYINT DEFAULT 0;
  DECLARE cur1 CURSOR FOR SELECT xname, id FROM table1;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

  OPEN cur1;
  read_loop: LOOP
    FETCH FROM cur1 INTO newname, xid;
    IF done THEN LEAVE read_loop; END IF;
    SELECT newname;
  END LOOP;
  CLOSE cur1;
END;
```

See also Section 25.8, “Restrictions on Stored Programs”.

13.6.5 Flow Control Statements

MySQL supports the `IF`, `CASE`, `ITERATE`, `LEAVE LOOP`, `WHILE`, and `REPEAT` constructs for flow control within stored programs. It also supports `RETURN` within stored functions.

Many of these constructs contain other statements, as indicated by the grammar specifications in the following sections. Such constructs may be nested. For example, an `IF` statement might contain a `WHILE` loop, which itself contains a `CASE` statement.

MySQL does not support `FOR` loops.

13.6.5.1 CASE Statement

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

Or:

```
CASE
    WHEN search_condition THEN statement_list
```

```
    [WHEN search_condition THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

The `CASE` statement for stored programs implements a complex conditional construct.

Note

There is also a `CASE` *operator*, which differs from the `CASE` *statement* described here. See Section 12.5, “Flow Control Functions”. The `CASE` statement cannot have an `ELSE NULL` clause, and it is terminated with `END CASE` instead of `END`.

For the first syntax, *`case_value`* is an expression. This value is compared to the *`when_value`* expression in each `WHEN` clause until one of them is equal. When an equal *`when_value`* is found, the corresponding `THEN` clause *`statement_list`* executes. If no *`when_value`* is equal, the `ELSE` clause *`statement_list`* executes, if there is one.

This syntax cannot be used to test for equality with `NULL` because `NULL = NULL` is false. See Section 3.3.4.6, “Working with NULL Values”.

For the second syntax, each `WHEN` clause *`search_condition`* expression is evaluated until one is true, at which point its corresponding `THEN` clause *`statement_list`* executes. If no *`search_condition`* is equal, the `ELSE` clause *`statement_list`* executes, if there is one.

If no *`when_value`* or *`search_condition`* matches the value tested and the `CASE` statement contains no `ELSE` clause, a `Case not found for CASE statement` error results.

Each *`statement_list`* consists of one or more SQL statements; an empty *`statement_list`* is not permitted.

To handle situations where no value is matched by any `WHEN` clause, use an `ELSE` containing an empty `BEGIN ... END` block, as shown in this example. (The indentation used here in the `ELSE` clause is for purposes of clarity only, and is not otherwise significant.)

```
DELIMITER |

CREATE PROCEDURE p()
  BEGIN
    DECLARE v INT DEFAULT 1;

    CASE v
      WHEN 2 THEN SELECT v;
      WHEN 3 THEN SELECT 0;
      ELSE
        BEGIN
        END;
    END CASE;
  END;
  |
```

13.6.5.2 IF Statement

```
IF search_condition THEN statement_list
    [ELSEIF search_condition THEN statement_list] ...
    [ELSE statement_list]
END IF
```

The `IF` statement for stored programs implements a basic conditional construct.

Note

There is also an `IF()` *function*, which differs from the `IF` *statement* described here. See Section 12.5, “Flow Control Functions”. The `IF` statement can have `THEN`, `ELSE`, and `ELSEIF` clauses, and it is terminated with `END IF`.

If a given *search_condition* evaluates to true, the corresponding THEN or ELSEIF clause *statement_list* executes. If no *search_condition* matches, the ELSE clause *statement_list* executes.

Each *statement_list* consists of one or more SQL statements; an empty *statement_list* is not permitted.

An IF ... END IF block, like all other flow-control blocks used within stored programs, must be terminated with a semicolon, as shown in this example:

```
DELIMITER //

CREATE FUNCTION SimpleCompare(n INT, m INT)
  RETURNS VARCHAR(20)

  BEGIN
    DECLARE s VARCHAR(20);

    IF n > m THEN SET s = '>';
    ELSEIF n = m THEN SET s = '=';
    ELSE SET s = '<';
    END IF;

    SET s = CONCAT(n, ' ', s, ' ', m);

    RETURN s;
  END //

DELIMITER ;
```

As with other flow-control constructs, IF ... END IF blocks may be nested within other flow-control constructs, including other IF statements. Each IF must be terminated by its own END IF followed by a semicolon. You can use indentation to make nested flow-control blocks more easily readable by humans (although this is not required by MySQL), as shown here:

```
DELIMITER //

CREATE FUNCTION VerboseCompare (n INT, m INT)
  RETURNS VARCHAR(50)

  BEGIN
    DECLARE s VARCHAR(50);

    IF n = m THEN SET s = 'equals';
    ELSE
      IF n > m THEN SET s = 'greater';
      ELSE SET s = 'less';
      END IF;

      SET s = CONCAT('is ', s, ' than');
    END IF;

    SET s = CONCAT(n, ' ', s, ' ', m, '.');

    RETURN s;
  END //

DELIMITER ;
```

In this example, the inner IF is evaluated only if n is not equal to m.

13.6.5.3 ITERATE Statement

```
ITERATE label
```

ITERATE can appear only within LOOP, REPEAT, and WHILE statements. ITERATE means “start the loop again.”

For an example, see Section 13.6.5.5, “LOOP Statement”.

13.6.5.4 LEAVE Statement

```
LEAVE label
```

This statement is used to exit the flow control construct that has the given label. If the label is for the outermost stored program block, `LEAVE` exits the program.

`LEAVE` can be used within `BEGIN ... END` or loop constructs (`LOOP`, `REPEAT`, `WHILE`).

For an example, see Section 13.6.5.5, “LOOP Statement”.

13.6.5.5 LOOP Statement

```
[begin_label:] LOOP
    statement_list
END LOOP [end_label]
```

`LOOP` implements a simple loop construct, enabling repeated execution of the statement list, which consists of one or more statements, each terminated by a semicolon (`;`) statement delimiter. The statements within the loop are repeated until the loop is terminated. Usually, this is accomplished with a `LEAVE` statement. Within a stored function, `RETURN` can also be used, which exits the function entirely.

Neglecting to include a loop-termination statement results in an infinite loop.

A `LOOP` statement can be labeled. For the rules regarding label use, see Section 13.6.2, “Statement Labels”.

Example:

```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
  label1: LOOP
    SET p1 = p1 + 1;
    IF p1 < 10 THEN
      ITERATE label1;
    END IF;
    LEAVE label1;
  END LOOP label1;
  SET @x = p1;
END;
```

13.6.5.6 REPEAT Statement

```
[begin_label:] REPEAT
    statement_list
UNTIL search_condition
END REPEAT [end_label]
```

The statement list within a `REPEAT` statement is repeated until the *`search_condition`* expression is true. Thus, a `REPEAT` always enters the loop at least once. *`statement_list`* consists of one or more statements, each terminated by a semicolon (`;`) statement delimiter.

A `REPEAT` statement can be labeled. For the rules regarding label use, see Section 13.6.2, “Statement Labels”.

Example:

```
mysql> delimiter //

mysql> CREATE PROCEDURE dorepeat(p1 INT)
       BEGIN
         SET @x = 0;
         REPEAT
           SET @x = @x + 1;
         UNTIL @x > p1 END REPEAT;
       END
       //
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CALL dorepeat(1000)//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x//
+------+
| @x   |
+------+
| 1001 |
+------+
1 row in set (0.00 sec)
```

13.6.5.7 RETURN Statement

```
RETURN expr
```

The `RETURN` statement terminates execution of a stored function and returns the value *`expr`* to the function caller. There must be at least one `RETURN` statement in a stored function. There may be more than one if the function has multiple exit points.

This statement is not used in stored procedures, triggers, or events. The `LEAVE` statement can be used to exit a stored program of those types.

13.6.5.8 WHILE Statement

```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```

The statement list within a `WHILE` statement is repeated as long as the *`search_condition`* expression is true. *`statement_list`* consists of one or more SQL statements, each terminated by a semicolon (`;`) statement delimiter.

A `WHILE` statement can be labeled. For the rules regarding label use, see Section 13.6.2, “Statement Labels”.

Example:

```
CREATE PROCEDURE dowhile()
BEGIN
  DECLARE v1 INT DEFAULT 5;

  WHILE v1 > 0 DO
    ...
    SET v1 = v1 - 1;
  END WHILE;
END;
```

13.6.6 Cursors

MySQL supports cursors inside stored programs. The syntax is as in embedded SQL. Cursors have these properties:

- Asensitive: The server may or may not make a copy of its result table
- Read only: Not updatable
- Nonscrollable: Can be traversed only in one direction and cannot skip rows

Cursor declarations must appear before handler declarations and after variable and condition declarations.

Example:

```
CREATE PROCEDURE curdemo()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE a CHAR(16);
  DECLARE b, c INT;
```

```
  DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
  DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN cur1;
  OPEN cur2;

  read_loop: LOOP
    FETCH cur1 INTO a, b;
    FETCH cur2 INTO c;
    IF done THEN
      LEAVE read_loop;
    END IF;
    IF b < c THEN
      INSERT INTO test.t3 VALUES (a,b);
    ELSE
      INSERT INTO test.t3 VALUES (a,c);
    END IF;
  END LOOP;

  CLOSE cur1;
  CLOSE cur2;
END;
```

13.6.6.1 Cursor CLOSE Statement

```
CLOSE cursor_name
```

This statement closes a previously opened cursor. For an example, see Section 13.6.6, “Cursors”.

An error occurs if the cursor is not open.

If not closed explicitly, a cursor is closed at the end of the `BEGIN ... END` block in which it was declared.

13.6.6.2 Cursor DECLARE Statement

```
DECLARE cursor_name CURSOR FOR select_statement
```

This statement declares a cursor and associates it with a `SELECT` statement that retrieves the rows to be traversed by the cursor. To fetch the rows later, use a `FETCH` statement. The number of columns retrieved by the `SELECT` statement must match the number of output variables specified in the `FETCH` statement.

The `SELECT` statement cannot have an `INTO` clause.

Cursor declarations must appear before handler declarations and after variable and condition declarations.

A stored program may contain multiple cursor declarations, but each cursor declared in a given block must have a unique name. For an example, see Section 13.6.6, “Cursors”.

For information available through `SHOW` statements, it is possible in many cases to obtain equivalent information by using a cursor with an `INFORMATION_SCHEMA` table.

13.6.6.3 Cursor FETCH Statement

```
FETCH [[NEXT] FROM] cursor_name INTO var_name [, var_name] ...
```

This statement fetches the next row for the `SELECT` statement associated with the specified cursor (which must be open), and advances the cursor pointer. If a row exists, the fetched columns are stored in the named variables. The number of columns retrieved by the `SELECT` statement must match the number of output variables specified in the `FETCH` statement.

If no more rows are available, a No Data condition occurs with SQLSTATE value `'02000'`. To detect this condition, you can set up a handler for it (or for a `NOT FOUND` condition). For an example, see Section 13.6.6, “Cursors”.

Be aware that another operation, such as a `SELECT` or another `FETCH`, may also cause the handler to execute by raising the same condition. If it is necessary to distinguish which operation raised the condition, place the operation within its own `BEGIN ... END` block so that it can be associated with its own handler.

13.6.6.4 Cursor OPEN Statement

```
OPEN cursor_name
```

This statement opens a previously declared cursor. For an example, see Section 13.6.6, “Cursors”.

13.6.6.5 Restrictions on Server-Side Cursors

Server-side cursors are implemented in the C API using the `mysql_stmt_attr_set()` function. The same implementation is used for cursors in stored routines. A server-side cursor enables a result set to be generated on the server side, but not transferred to the client except for those rows that the client requests. For example, if a client executes a query but is only interested in the first row, the remaining rows are not transferred.

In MySQL, a server-side cursor is materialized into an internal temporary table. Initially, this is a `MEMORY` table, but is converted to a `MyISAM` table when its size exceeds the minimum value of the `max_heap_table_size` and `tmp_table_size` system variables. The same restrictions apply to internal temporary tables created to hold the result set for a cursor as for other uses of internal temporary tables. See Section 8.4.4, “Internal Temporary Table Use in MySQL”. One limitation of the implementation is that for a large result set, retrieving its rows through a cursor might be slow.

Cursors are read only; you cannot use a cursor to update rows.

`UPDATE WHERE CURRENT OF` and `DELETE WHERE CURRENT OF` are not implemented, because updatable cursors are not supported.

Cursors are nonholdable (not held open after a commit).

Cursors are asensitive.

Cursors are nonscrollable.

Cursors are not named. The statement handler acts as the cursor ID.

You can have open only a single cursor per prepared statement. If you need several cursors, you must prepare several statements.

You cannot use a cursor for a statement that generates a result set if the statement is not supported in prepared mode. This includes statements such as `CHECK TABLE`, `HANDLER READ`, and `SHOW BINLOG EVENTS`.

13.6.7 Condition Handling

Conditions may arise during stored program execution that require special handling, such as exiting the current program block or continuing execution. Handlers can be defined for general conditions such as warnings or exceptions, or for specific conditions such as a particular error code. Specific conditions can be assigned names and referred to that way in handlers.

To name a condition, use the `DECLARE ... CONDITION` statement. To declare a handler, use the `DECLARE ... HANDLER` statement. See Section 13.6.7.1, “DECLARE ... CONDITION Statement”, and Section 13.6.7.2, “DECLARE ... HANDLER Statement”. For information about how the server chooses handlers when a condition occurs, see Section 13.6.7.6, “Scope Rules for Handlers”.

To raise a condition, use the `SIGNAL` statement. To modify condition information within a condition handler, use `RESIGNAL`. See Section 13.6.7.1, “DECLARE ... CONDITION Statement”, and Section 13.6.7.2, “DECLARE ... HANDLER Statement”.

To retrieve information from the diagnostics area, use the `GET DIAGNOSTICS` statement (see Section 13.6.7.3, “GET DIAGNOSTICS Statement”). For information about the diagnostics area, see Section 13.6.7.7, “The MySQL Diagnostics Area”.

13.6.7.1 DECLARE ... CONDITION Statement

```
DECLARE condition_name CONDITION FOR condition_value

condition_value: {
    mysql_error_code
  | SQLSTATE [VALUE] sqlstate_value
}
```

The `DECLARE ... CONDITION` statement declares a named error condition, associating a name with a condition that needs specific handling. The name can be referred to in a subsequent `DECLARE ... HANDLER` statement (see Section 13.6.7.2, “DECLARE ... HANDLER Statement”).

Condition declarations must appear before cursor or handler declarations.

The *`condition_value`* for `DECLARE ... CONDITION` indicates the specific condition or class of conditions to associate with the condition name. It can take the following forms:

- *`mysql_error_code`*: An integer literal indicating a MySQL error code.

  Do not use MySQL error code 0 because that indicates success rather than an error condition. For a list of MySQL error codes, see Server Error Message Reference.

- SQLSTATE [VALUE] *`sqlstate_value`*: A 5-character string literal indicating an SQLSTATE value.

  Do not use SQLSTATE values that begin with `'00'` because those indicate success rather than an error condition. For a list of SQLSTATE values, see Server Error Message Reference.

Condition names referred to in `SIGNAL` or use `RESIGNAL` statements must be associated with SQLSTATE values, not MySQL error codes.

Using names for conditions can help make stored program code clearer. For example, this handler applies to attempts to drop a nonexistent table, but that is apparent only if you know that 1051 is the MySQL error code for “unknown table”:

```
DECLARE CONTINUE HANDLER FOR 1051
  BEGIN
    -- body of handler
  END;
```

By declaring a name for the condition, the purpose of the handler is more readily seen:

```
DECLARE no_such_table CONDITION FOR 1051;
DECLARE CONTINUE HANDLER FOR no_such_table
  BEGIN
    -- body of handler
  END;
```

Here is a named condition for the same condition, but based on the corresponding SQLSTATE value rather than the MySQL error code:

```
DECLARE no_such_table CONDITION FOR SQLSTATE '42S02';
DECLARE CONTINUE HANDLER FOR no_such_table
  BEGIN
    -- body of handler
  END;
```

13.6.7.2 DECLARE ... HANDLER Statement

```
DECLARE handler_action HANDLER
    FOR condition_value [, condition_value] ...
    statement
```

```
handler_action: {
    CONTINUE
  | EXIT
  | UNDO
}

condition_value: {
    mysql_error_code
  | SQLSTATE [VALUE] sqlstate_value
  | condition_name
  | SQLWARNING
  | NOT FOUND
  | SQLEXCEPTION
}
```

The `DECLARE ... HANDLER` statement specifies a handler that deals with one or more conditions. If one of these conditions occurs, the specified *`statement`* executes. *`statement`* can be a simple statement such as `SET var_name = value`, or a compound statement written using `BEGIN` and `END` (see Section 13.6.1, “BEGIN ... END Compound Statement”).

Handler declarations must appear after variable or condition declarations.

The *`handler_action`* value indicates what action the handler takes after execution of the handler statement:

- `CONTINUE`: Execution of the current program continues.
- `EXIT`: Execution terminates for the `BEGIN ... END` compound statement in which the handler is declared. This is true even if the condition occurs in an inner block.
- `UNDO`: Not supported.

The *`condition_value`* for `DECLARE ... HANDLER` indicates the specific condition or class of conditions that activates the handler. It can take the following forms:

- *`mysql_error_code`*: An integer literal indicating a MySQL error code, such as 1051 to specify “unknown table”:

  ```
  DECLARE CONTINUE HANDLER FOR 1051
    BEGIN
      -- body of handler
    END;
  ```

  Do not use MySQL error code 0 because that indicates success rather than an error condition. For a list of MySQL error codes, see Server Error Message Reference.

- SQLSTATE [VALUE] *`sqlstate_value`*: A 5-character string literal indicating an SQLSTATE value, such as `'42S01'` to specify “unknown table”:

  ```
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'
    BEGIN
      -- body of handler
    END;
  ```

  Do not use SQLSTATE values that begin with `'00'` because those indicate success rather than an error condition. For a list of SQLSTATE values, see Server Error Message Reference.

- *`condition_name`*: A condition name previously specified with `DECLARE ... CONDITION`. A condition name can be associated with a MySQL error code or SQLSTATE value. See Section 13.6.7.1, “DECLARE ... CONDITION Statement”.

- `SQLWARNING`: Shorthand for the class of SQLSTATE values that begin with `'01'`.

  ```
  DECLARE CONTINUE HANDLER FOR SQLWARNING
    BEGIN
      -- body of handler
    END;
  ```

- `NOT FOUND`: Shorthand for the class of SQLSTATE values that begin with `'02'`. This is relevant within the context of cursors and is used to control what happens when a cursor reaches the end of a data set. If no more rows are available, a No Data condition occurs with SQLSTATE value `'02000'`. To detect this condition, you can set up a handler for it or for a `NOT FOUND` condition.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND
  BEGIN
    -- body of handler
  END;
```

  For another example, see Section 13.6.6, “Cursors”. The `NOT FOUND` condition also occurs for `SELECT ... INTO var_list` statements that retrieve no rows.

- `SQLEXCEPTION`: Shorthand for the class of SQLSTATE values that do not begin with `'00'`, `'01'`, or `'02'`.

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
  BEGIN
    -- body of handler
  END;
```

For information about how the server chooses handlers when a condition occurs, see Section 13.6.7.6, “Scope Rules for Handlers”.

If a condition occurs for which no handler has been declared, the action taken depends on the condition class:

- For `SQLEXCEPTION` conditions, the stored program terminates at the statement that raised the condition, as if there were an `EXIT` handler. If the program was called by another stored program, the calling program handles the condition using the handler selection rules applied to its own handlers.
- For `SQLWARNING` conditions, the program continues executing, as if there were a `CONTINUE` handler.
- For `NOT FOUND` conditions, if the condition was raised normally, the action is `CONTINUE`. If it was raised by `SIGNAL` or `RESIGNAL`, the action is `EXIT`.

The following example uses a handler for `SQLSTATE '23000'`, which occurs for a duplicate-key error:

```
mysql> CREATE TABLE test.t (s1 INT, PRIMARY KEY (s1));
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter //

mysql> CREATE PROCEDURE handlerdemo ()
       BEGIN
         DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;
         SET @x = 1;
         INSERT INTO test.t VALUES (1);
         SET @x = 2;
         INSERT INTO test.t VALUES (1);
         SET @x = 3;
       END;
       //
Query OK, 0 rows affected (0.00 sec)

mysql> CALL handlerdemo()//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x//
    +------+
    | @x   |
    +------+
    | 3    |
    +------+
    1 row in set (0.00 sec)
```

Notice that `@x` is `3` after the procedure executes, which shows that execution continued to the end of the procedure after the error occurred. If the `DECLARE ... HANDLER` statement had not been present, MySQL would have taken the default action (`EXIT`) after the second `INSERT` failed due to the `PRIMARY KEY` constraint, and `SELECT @x` would have returned `2`.

To ignore a condition, declare a `CONTINUE` handler for it and associate it with an empty block. For example:

```
DECLARE CONTINUE HANDLER FOR SQLWARNING BEGIN END;
```

The scope of a block label does not include the code for handlers declared within the block. Therefore, the statement associated with a handler cannot use `ITERATE` or `LEAVE` to refer to labels for blocks that enclose the handler declaration. Consider the following example, where the `REPEAT` block has a label of `retry`:

```
CREATE PROCEDURE p ()
BEGIN
  DECLARE i INT DEFAULT 3;
  retry:
    REPEAT
      BEGIN
        DECLARE CONTINUE HANDLER FOR SQLWARNING
          BEGIN
            ITERATE retry;    # illegal
          END;
        IF i < 0 THEN
          LEAVE retry;        # legal
        END IF;
        SET i = i - 1;
      END;
    UNTIL FALSE END REPEAT;
END;
```

The `retry` label is in scope for the `IF` statement within the block. It is not in scope for the `CONTINUE` handler, so the reference there is invalid and results in an error:

```
ERROR 1308 (42000): LEAVE with no matching label: retry
```

To avoid references to outer labels in handlers, use one of these strategies:

- To leave the block, use an `EXIT` handler. If no block cleanup is required, the `BEGIN ... END` handler body can be empty:

  ```
  DECLARE EXIT HANDLER FOR SQLWARNING BEGIN END;
  ```

  Otherwise, put the cleanup statements in the handler body:

  ```
  DECLARE EXIT HANDLER FOR SQLWARNING
    BEGIN
      block cleanup statements
    END;
  ```

- To continue execution, set a status variable in a `CONTINUE` handler that can be checked in the enclosing block to determine whether the handler was invoked. The following example uses the variable `done` for this purpose:

  ```
  CREATE PROCEDURE p ()
  BEGIN
    DECLARE i INT DEFAULT 3;
    DECLARE done INT DEFAULT FALSE;
    retry:
      REPEAT
        BEGIN
          DECLARE CONTINUE HANDLER FOR SQLWARNING
            BEGIN
              SET done = TRUE;
            END;
          IF done OR i < 0 THEN
  ```

```
          LEAVE retry;
        END IF;
        SET i = i - 1;
      END;
    UNTIL FALSE END REPEAT;
END;
```

13.6.7.3 GET DIAGNOSTICS Statement

```
GET [CURRENT | STACKED] DIAGNOSTICS {
    statement_information_item
    [, statement_information_item] ...
  | CONDITION condition_number
    condition_information_item
    [, condition_information_item] ...
}

statement_information_item:
    target = statement_information_item_name

condition_information_item:
    target = condition_information_item_name

statement_information_item_name: {
    NUMBER
  | ROW_COUNT
}

condition_information_item_name: {
    CLASS_ORIGIN
  | SUBCLASS_ORIGIN
  | RETURNED_SQLSTATE
  | MESSAGE_TEXT
  | MYSQL_ERRNO
  | CONSTRAINT_CATALOG
  | CONSTRAINT_SCHEMA
  | CONSTRAINT_NAME
  | CATALOG_NAME
  | SCHEMA_NAME
  | TABLE_NAME
  | COLUMN_NAME
  | CURSOR_NAME
}

condition_number, target:
    (see following discussion)
```

SQL statements produce diagnostic information that populates the diagnostics area. The `GET DIAGNOSTICS` statement enables applications to inspect this information. (You can also use `SHOW WARNINGS` or `SHOW ERRORS` to see conditions or errors.)

No special privileges are required to execute `GET DIAGNOSTICS`.

The keyword `CURRENT` means to retrieve information from the current diagnostics area. The keyword `STACKED` means to retrieve information from the second diagnostics area, which is available only if the current context is a condition handler. If neither keyword is given, the default is to use the current diagnostics area.

The `GET DIAGNOSTICS` statement is typically used in a handler within a stored program. It is a MySQL extension that `GET [CURRENT] DIAGNOSTICS` is permitted outside handler context to check the execution of any SQL statement. For example, if you invoke the `mysql` client program, you can enter these statements at the prompt:

```
mysql> DROP TABLE test.no_such_table;
ERROR 1051 (42S02): Unknown table 'test.no_such_table'
mysql> GET DIAGNOSTICS CONDITION 1
        @p1 = RETURNED_SQLSTATE, @p2 = MESSAGE_TEXT;
mysql> SELECT @p1, @p2;
+-------+----------------------------------+
```

```
| @p1   | @p2                                 |
+-------+-------------------------------------+
| 42S02 | Unknown table 'test.no_such_table' |
+-------+-------------------------------------+
```

This extension applies only to the current diagnostics area. It does not apply to the second diagnostics area because `GET STACKED DIAGNOSTICS` is permitted only if the current context is a condition handler. If that is not the case, a `GET STACKED DIAGNOSTICS when handler not active` error occurs.

For a description of the diagnostics area, see Section 13.6.7.7, “The MySQL Diagnostics Area”. Briefly, it contains two kinds of information:

- Statement information, such as the number of conditions that occurred or the affected-rows count.
- Condition information, such as the error code and message. If a statement raises multiple conditions, this part of the diagnostics area has a condition area for each one. If a statement raises no conditions, this part of the diagnostics area is empty.

For a statement that produces three conditions, the diagnostics area contains statement and condition information like this:

```
Statement information:
  row count
  ... other statement information items ...
Condition area list:
  Condition area 1:
    error code for condition 1
    error message for condition 1
    ... other condition information items ...
  Condition area 2:
    error code for condition 2:
    error message for condition 2
    ... other condition information items ...
  Condition area 3:
    error code for condition 3
    error message for condition 3
    ... other condition information items ...
```

`GET DIAGNOSTICS` can obtain either statement or condition information, but not both in the same statement:

- To obtain statement information, retrieve the desired statement items into target variables. This instance of `GET DIAGNOSTICS` assigns the number of available conditions and the rows-affected count to the user variables `@p1` and `@p2`:

  ```
  GET DIAGNOSTICS @p1 = NUMBER, @p2 = ROW_COUNT;
  ```

- To obtain condition information, specify the condition number and retrieve the desired condition items into target variables. This instance of `GET DIAGNOSTICS` assigns the SQLSTATE value and error message to the user variables `@p3` and `@p4`:

  ```
  GET DIAGNOSTICS CONDITION 1
    @p3 = RETURNED_SQLSTATE, @p4 = MESSAGE_TEXT;
  ```

The retrieval list specifies one or more *`target = item_name`* assignments, separated by commas. Each assignment names a target variable and either a *`statement_information_item_name`* or *`condition_information_item_name`* designator, depending on whether the statement retrieves statement or condition information.

Valid *`target`* designators for storing item information can be stored procedure or function parameters, stored program local variables declared with `DECLARE`, or user-defined variables.

Valid *`condition_number`* designators can be stored procedure or function parameters, stored program local variables declared with `DECLARE`, user-defined variables, system variables, or literals. A character literal may include a *`_charset`* introducer. A warning occurs if the condition number is not

in the range from 1 to the number of condition areas that have information. In this case, the warning is added to the diagnostics area without clearing it.

When a condition occurs, MySQL does not populate all condition items recognized by `GET DIAGNOSTICS`. For example:

```
mysql> GET DIAGNOSTICS CONDITION 1
         @p5 = SCHEMA_NAME, @p6 = TABLE_NAME;
mysql> SELECT @p5, @p6;
+------+------+
| @p5  | @p6  |
+------+------+
|      |      |
+------+------+
```

In standard SQL, if there are multiple conditions, the first condition relates to the `SQLSTATE` value returned for the previous SQL statement. In MySQL, this is not guaranteed. To get the main error, you cannot do this:

```
GET DIAGNOSTICS CONDITION 1 @errno = MYSQL_ERRNO;
```

Instead, retrieve the condition count first, then use it to specify which condition number to inspect:

```
GET DIAGNOSTICS @cno = NUMBER;
GET DIAGNOSTICS CONDITION @cno @errno = MYSQL_ERRNO;
```

For information about permissible statement and condition information items, and which ones are populated when a condition occurs, see Diagnostics Area Information Items.

Here is an example that uses `GET DIAGNOSTICS` and an exception handler in stored procedure context to assess the outcome of an insert operation. If the insert was successful, the procedure uses `GET DIAGNOSTICS` to get the rows-affected count. This shows that you can use `GET DIAGNOSTICS` multiple times to retrieve information about a statement as long as the current diagnostics area has not been cleared.

```
CREATE PROCEDURE do_insert(value INT)
BEGIN
  -- Declare variables to hold diagnostics area information
  DECLARE code CHAR(5) DEFAULT '00000';
  DECLARE msg TEXT;
  DECLARE nrows INT;
  DECLARE result TEXT;
  -- Declare exception handler for failed insert
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    BEGIN
      GET DIAGNOSTICS CONDITION 1
        code = RETURNED_SQLSTATE, msg = MESSAGE_TEXT;
    END;

  -- Perform the insert
  INSERT INTO t1 (int_col) VALUES(value);
  -- Check whether the insert was successful
  IF code = '00000' THEN
    GET DIAGNOSTICS nrows = ROW_COUNT;
    SET result = CONCAT('insert succeeded, row count = ',nrows);
  ELSE
    SET result = CONCAT('insert failed, error = ',code,', message = ',msg);
  END IF;
  -- Say what happened
  SELECT result;
END;
```

Suppose that `t1.int_col` is an integer column that is declared as `NOT NULL`. The procedure produces these results when invoked to insert non-`NULL` and `NULL` values, respectively:

```
mysql> CALL do_insert(1);
+---------------------------------+
| result                          |
+---------------------------------+
```

```
| insert succeeded, row count = 1 |
+---------------------------------+

mysql> CALL do_insert(NULL);
+-------------------------------------------------------------------------+
| result                                                                  |
+-------------------------------------------------------------------------+
| insert failed, error = 23000, message = Column 'int_col' cannot be null |
+-------------------------------------------------------------------------+
```

When a condition handler activates, a push to the diagnostics area stack occurs:

- The first (current) diagnostics area becomes the second (stacked) diagnostics area and a new current diagnostics area is created as a copy of it.
- `GET [CURRENT] DIAGNOSTICS` and `GET STACKED DIAGNOSTICS` can be used within the handler to access the contents of the current and stacked diagnostics areas.
- Initially, both diagnostics areas return the same result, so it is possible to get information from the current diagnostics area about the condition that activated the handler, *as long as* you execute no statements within the handler that change its current diagnostics area.
- However, statements executing within the handler can modify the current diagnostics area, clearing and setting its contents according to the normal rules (see How the Diagnostics Area is Cleared and Populated).

  A more reliable way to obtain information about the handler-activating condition is to use the stacked diagnostics area, which cannot be modified by statements executing within the handler except `RESIGNAL`. For information about when the current diagnostics area is set and cleared, see Section 13.6.7.7, “The MySQL Diagnostics Area”.

The next example shows how `GET STACKED DIAGNOSTICS` can be used within a handler to obtain information about the handled exception, even after the current diagnostics area has been modified by handler statements.

Within a stored procedure `p()`, we attempt to insert two values into a table that contains a `TEXT NOT NULL` column. The first value is a non-`NULL` string and the second is `NULL`. The column prohibits `NULL` values, so the first insert succeeds but the second causes an exception. The procedure includes an exception handler that maps attempts to insert `NULL` into inserts of the empty string:

```
DROP TABLE IF EXISTS t1;
CREATE TABLE t1 (c1 TEXT NOT NULL);
DROP PROCEDURE IF EXISTS p;
delimiter //
CREATE PROCEDURE p ()
BEGIN
  -- Declare variables to hold diagnostics area information
  DECLARE errcount INT;
  DECLARE errno INT;
  DECLARE msg TEXT;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    -- Here the current DA is nonempty because no prior statements
    -- executing within the handler have cleared it
    GET CURRENT DIAGNOSTICS CONDITION 1
      errno = MYSQL_ERRNO, msg = MESSAGE_TEXT;
    SELECT 'current DA before mapped insert' AS op, errno, msg;
    GET STACKED DIAGNOSTICS CONDITION 1
      errno = MYSQL_ERRNO, msg = MESSAGE_TEXT;
    SELECT 'stacked DA before mapped insert' AS op, errno, msg;

    -- Map attempted NULL insert to empty string insert
    INSERT INTO t1 (c1) VALUES('');

    -- Here the current DA should be empty (if the INSERT succeeded),
    -- so check whether there are conditions before attempting to
    -- obtain condition information
    GET CURRENT DIAGNOSTICS errcount = NUMBER;
```

```
    IF errcount = 0
    THEN
      SELECT 'mapped insert succeeded, current DA is empty' AS op;
    ELSE
      GET CURRENT DIAGNOSTICS CONDITION 1
        errno = MYSQL_ERRNO, msg = MESSAGE_TEXT;
      SELECT 'current DA after mapped insert' AS op, errno, msg;
    END IF ;
    GET STACKED DIAGNOSTICS CONDITION 1
      errno = MYSQL_ERRNO, msg = MESSAGE_TEXT;
    SELECT 'stacked DA after mapped insert' AS op, errno, msg;
  END;
  INSERT INTO t1 (c1) VALUES('string 1');
  INSERT INTO t1 (c1) VALUES(NULL);
END;
//
delimiter ;
CALL p();
SELECT * FROM t1;
```

When the handler activates, a copy of the current diagnostics area is pushed to the diagnostics area stack. The handler first displays the contents of the current and stacked diagnostics areas, which are both the same initially:

```
+---------------------------------+-------+----------------------------+
| op                              | errno | msg                        |
+---------------------------------+-------+----------------------------+
| current DA before mapped insert |  1048 | Column 'c1' cannot be null |
+---------------------------------+-------+----------------------------+

+---------------------------------+-------+----------------------------+
| op                              | errno | msg                        |
+---------------------------------+-------+----------------------------+
| stacked DA before mapped insert |  1048 | Column 'c1' cannot be null |
+---------------------------------+-------+----------------------------+
```

Statements executing after the `GET DIAGNOSTICS` statements may reset the current diagnostics area. statements may reset the current diagnostics area. For example, the handler maps the `NULL` insert to an empty-string insert and displays the result. The new insert succeeds and clears the current diagnostics area, but the stacked diagnostics area remains unchanged and still contains information about the condition that activated the handler:

```
+--------------------------------------------+
| op                                         |
+--------------------------------------------+
| mapped insert succeeded, current DA is empty |
+--------------------------------------------+

+--------------------------------+-------+----------------------------+
| op                             | errno | msg                        |
+--------------------------------+-------+----------------------------+
| stacked DA after mapped insert |  1048 | Column 'c1' cannot be null |
+--------------------------------+-------+----------------------------+
```

When the condition handler ends, its current diagnostics area is popped from the stack and the stacked diagnostics area becomes the current diagnostics area in the stored procedure.

After the procedure returns, the table contains two rows. The empty row results from the attempt to insert `NULL` that was mapped to an empty-string insert:

```
+----------+
| c1       |
+----------+
| string 1 |
|          |
+----------+
```

In the preceding example, the first two `GET DIAGNOSTICS` statements within the condition handler that retrieve information from the current and stacked diagnostics areas return the same values. This

is not the case if statements that reset the current diagnostics area execute earlier within the handler. Suppose that `p()` is rewritten to place the `DECLARE` statements within the handler definition rather than preceding it:

```
CREATE PROCEDURE p ()
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    -- Declare variables to hold diagnostics area information
    DECLARE errcount INT;
    DECLARE errno INT;
    DECLARE msg TEXT;
    GET CURRENT DIAGNOSTICS CONDITION 1
      errno = MYSQL_ERRNO, msg = MESSAGE_TEXT;
    SELECT 'current DA before mapped insert' AS op, errno, msg;
    GET STACKED DIAGNOSTICS CONDITION 1
      errno = MYSQL_ERRNO, msg = MESSAGE_TEXT;
    SELECT 'stacked DA before mapped insert' AS op, errno, msg;
...
```

In this case, the result is version dependent:

- Before MySQL 5.7.2, `DECLARE` does not change the current diagnostics area, so the first two `GET DIAGNOSTICS` statements return the same result, just as in the original version of `p()`.

  In MySQL 5.7.2, work was done to ensure that all nondiagnostic statements populate the diagnostics area, per the SQL standard. `DECLARE` is one of them, so in 5.7.2 and higher, `DECLARE` statements executing at the beginning of the handler clear the current diagnostics area and the `GET DIAGNOSTICS` statements produce different results:

```
+---------------------------------+-------+------+
| op                              | errno | msg  |
+---------------------------------+-------+------+
| current DA before mapped insert |  NULL | NULL |
+---------------------------------+-------+------+

+---------------------------------+-------+---------------------------+
| op                              | errno | msg                       |
+---------------------------------+-------+---------------------------+
| stacked DA before mapped insert |  1048 | Column 'c1' cannot be null |
+---------------------------------+-------+---------------------------+
```

To avoid this issue within a condition handler when seeking to obtain information about the condition that activated the handler, be sure to access the stacked diagnostics area, not the current diagnostics area.

13.6.7.4 RESIGNAL Statement

```
RESIGNAL [condition_value]
    [SET signal_information_item
    [, signal_information_item] ...]

condition_value: {
    SQLSTATE [VALUE] sqlstate_value
  | condition_name
}

signal_information_item:
    condition_information_item_name = simple_value_specification

condition_information_item_name: {
    CLASS_ORIGIN
  | SUBCLASS_ORIGIN
  | MESSAGE_TEXT
  | MYSQL_ERRNO
  | CONSTRAINT_CATALOG
  | CONSTRAINT_SCHEMA
  | CONSTRAINT_NAME
  | CATALOG_NAME
```

```
  | SCHEMA_NAME
  | TABLE_NAME
  | COLUMN_NAME
  | CURSOR_NAME
}

condition_name, simple_value_specification:
    (see following discussion)
```

RESIGNAL passes on the error condition information that is available during execution of a condition handler within a compound statement inside a stored procedure or function, trigger, or event. RESIGNAL may change some or all information before passing it on. RESIGNAL is related to SIGNAL, but instead of originating a condition as SIGNAL does, RESIGNAL relays existing condition information, possibly after modifying it.

RESIGNAL makes it possible to both handle an error and return the error information. Otherwise, by executing an SQL statement within the handler, information that caused the handler's activation is destroyed. RESIGNAL also can make some procedures shorter if a given handler can handle part of a situation, then pass the condition “up the line” to another handler.

No privileges are required to execute the RESIGNAL statement.

All forms of RESIGNAL require that the current context be a condition handler. Otherwise, RESIGNAL is illegal and a RESIGNAL when handler not active error occurs.

To retrieve information from the diagnostics area, use the GET DIAGNOSTICS statement (see Section 13.6.7.3, “GET DIAGNOSTICS Statement”). For information about the diagnostics area, see Section 13.6.7.7, “The MySQL Diagnostics Area”.

- RESIGNAL Overview
- RESIGNAL Alone
- RESIGNAL with New Signal Information
- RESIGNAL with a Condition Value and Optional New Signal Information
- RESIGNAL Requires Condition Handler Context

RESIGNAL Overview

For *condition_value* and *signal_information_item*, the definitions and rules are the same for RESIGNAL as for SIGNAL. For example, the *condition_value* can be an SQLSTATE value, and the value can indicate errors, warnings, or “not found.” For additional information, see Section 13.6.7.5, “SIGNAL Statement”.

The RESIGNAL statement takes *condition_value* and SET clauses, both of which are optional. This leads to several possible uses:

- RESIGNAL alone:

```
RESIGNAL;
```

- RESIGNAL with new signal information:

```
RESIGNAL SET signal_information_item [, signal_information_item] ...;
```

- RESIGNAL with a condition value and possibly new signal information:

```
RESIGNAL condition_value
    [SET signal_information_item [, signal_information_item] ...];
```

These use cases all cause changes to the diagnostics and condition areas:

- A diagnostics area contains one or more condition areas.

- A condition area contains condition information items, such as the `SQLSTATE` value, `MYSQL_ERRNO`, or `MESSAGE_TEXT`.

There is a stack of diagnostics areas. When a handler takes control, it pushes a diagnostics area to the top of the stack, so there are two diagnostics areas during handler execution:

- The first (current) diagnostics area, which starts as a copy of the last diagnostics area, but is overwritten by the first statement in the handler that changes the current diagnostics area.
- The last (stacked) diagnostics area, which has the condition areas that were set up before the handler took control.

The maximum number of condition areas in a diagnostics area is determined by the value of the `max_error_count` system variable. See Diagnostics Area-Related System Variables.

RESIGNAL Alone

A simple `RESIGNAL` alone means “pass on the error with no change.” It restores the last diagnostics area and makes it the current diagnostics area. That is, it “pops” the diagnostics area stack.

Within a condition handler that catches a condition, one use for `RESIGNAL` alone is to perform some other actions, and then pass on without change the original condition information (the information that existed before entry into the handler).

Example:

```
DROP TABLE IF EXISTS xx;
delimiter //
CREATE PROCEDURE p ()
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    SET @error_count = @error_count + 1;
    IF @a = 0 THEN RESIGNAL; END IF;
  END;
  DROP TABLE xx;
END//
delimiter ;
SET @error_count = 0;
SET @a = 0;
CALL p();
```

Suppose that the `DROP TABLE xx` statement fails. The diagnostics area stack looks like this:

```
DA 1. ERROR 1051 (42S02): Unknown table 'xx'
```

Then execution enters the `EXIT` handler. It starts by pushing a diagnostics area to the top of the stack, which now looks like this:

```
DA 1. ERROR 1051 (42S02): Unknown table 'xx'
DA 2. ERROR 1051 (42S02): Unknown table 'xx'
```

At this point, the contents of the first (current) and second (stacked) diagnostics areas are the same. The first diagnostics area may be modified by statements executing subsequently within the handler.

Usually a procedure statement clears the first diagnostics area. `BEGIN` is an exception, it does not clear, it does nothing. `SET` is not an exception, it clears, performs the operation, and produces a result of “success.” The diagnostics area stack now looks like this:

```
DA 1. ERROR 0000 (00000): Successful operation
DA 2. ERROR 1051 (42S02): Unknown table 'xx'
```

At this point, if `@a = 0`, `RESIGNAL` pops the diagnostics area stack, which now looks like this:

```
DA 1. ERROR 1051 (42S02): Unknown table 'xx'
```

And that is what the caller sees.

If @a is not 0, the handler simply ends, which means that there is no more use for the current diagnostics area (it has been “handled”), so it can be thrown away, causing the stacked diagnostics area to become the current diagnostics area again. The diagnostics area stack looks like this:

```
DA 1. ERROR 0000 (00000): Successful operation
```

The details make it look complex, but the end result is quite useful: Handlers can execute without destroying information about the condition that caused activation of the handler.

RESIGNAL with New Signal Information

RESIGNAL with a SET clause provides new signal information, so the statement means “pass on the error with changes”:

```
RESIGNAL SET signal_information_item [, signal_information_item] ...;
```

As with RESIGNAL alone, the idea is to pop the diagnostics area stack so that the original information goes out. Unlike RESIGNAL alone, anything specified in the SET clause changes.

Example:

```
DROP TABLE IF EXISTS xx;
delimiter //
CREATE PROCEDURE p ()
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    SET @error_count = @error_count + 1;
    IF @a = 0 THEN RESIGNAL SET MYSQL_ERRNO = 5; END IF;
  END;
  DROP TABLE xx;
END//
delimiter ;
SET @error_count = 0;
SET @a = 0;
CALL p();
```

Remember from the previous discussion that RESIGNAL alone results in a diagnostics area stack like this:

```
DA 1. ERROR 1051 (42S02): Unknown table 'xx'
```

The RESIGNAL SET MYSQL_ERRNO = 5 statement results in this stack instead, which is what the caller sees:

```
DA 1. ERROR 5 (42S02): Unknown table 'xx'
```

In other words, it changes the error number, and nothing else.

The RESIGNAL statement can change any or all of the signal information items, making the first condition area of the diagnostics area look quite different.

RESIGNAL with a Condition Value and Optional New Signal Information

RESIGNAL with a condition value means “push a condition into the current diagnostics area.” If the SET clause is present, it also changes the error information.

```
RESIGNAL condition_value
    [SET signal_information_item [, signal_information_item] ...];
```

This form of RESIGNAL restores the last diagnostics area and makes it the current diagnostics area. That is, it “pops” the diagnostics area stack, which is the same as what a simple RESIGNAL alone would do. However, it also changes the diagnostics area depending on the condition value or signal information.

Example:

```
DROP TABLE IF EXISTS xx;
delimiter //
CREATE PROCEDURE p ()
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    SET @error_count = @error_count + 1;
    IF @a = 0 THEN RESIGNAL SQLSTATE '45000' SET MYSQL_ERRNO=5; END IF;
  END;
  DROP TABLE xx;
END//
delimiter ;
SET @error_count = 0;
SET @a = 0;
SET @@max_error_count = 2;
CALL p();
SHOW ERRORS;
```

This is similar to the previous example, and the effects are the same, except that if `RESIGNAL` happens, the current condition area looks different at the end. (The reason the condition adds to rather than replaces the existing condition is the use of a condition value.)

The `RESIGNAL` statement includes a condition value (`SQLSTATE '45000'`), so it adds a new condition area, resulting in a diagnostics area stack that looks like this:

```
DA 1. (condition 2) ERROR 1051 (42S02): Unknown table 'xx'
      (condition 1) ERROR 5 (45000) Unknown table 'xx'
```

The result of `CALL p()` and `SHOW ERRORS` for this example is:

```
mysql> CALL p();
ERROR 5 (45000): Unknown table 'xx'
mysql> SHOW ERRORS;
+-------+------+---------------------------------+
| Level | Code | Message                         |
+-------+------+---------------------------------+
| Error | 1051 | Unknown table 'xx'              |
| Error |    5 | Unknown table 'xx'              |
+-------+------+---------------------------------+
```

RESIGNAL Requires Condition Handler Context

All forms of `RESIGNAL` require that the current context be a condition handler. Otherwise, `RESIGNAL` is illegal and a `RESIGNAL when handler not active` error occurs. For example:

```
mysql> CREATE PROCEDURE p () RESIGNAL;
Query OK, 0 rows affected (0.00 sec)

mysql> CALL p();
ERROR 1645 (0K000): RESIGNAL when handler not active
```

Here is a more difficult example:

```
delimiter //
CREATE FUNCTION f () RETURNS INT
BEGIN
  RESIGNAL;
  RETURN 5;
END//
CREATE PROCEDURE p ()
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION SET @a=f();
  SIGNAL SQLSTATE '55555';
END//
delimiter ;
CALL p();
```

`RESIGNAL` occurs within the stored function `f()`. Although `f()` itself is invoked within the context of the `EXIT` handler, execution within `f()` has its own context, which is not handler context. Thus, `RESIGNAL` within `f()` results in a “handler not active” error.

13.6.7.5 SIGNAL Statement

```
SIGNAL condition_value
    [SET signal_information_item
    [, signal_information_item] ...]

condition_value: {
    SQLSTATE [VALUE] sqlstate_value
  | condition_name
}

signal_information_item:
    condition_information_item_name = simple_value_specification

condition_information_item_name: {
    CLASS_ORIGIN
  | SUBCLASS_ORIGIN
  | MESSAGE_TEXT
  | MYSQL_ERRNO
  | CONSTRAINT_CATALOG
  | CONSTRAINT_SCHEMA
  | CONSTRAINT_NAME
  | CATALOG_NAME
  | SCHEMA_NAME
  | TABLE_NAME
  | COLUMN_NAME
  | CURSOR_NAME
}

condition_name, simple_value_specification:
    (see following discussion)
```

`SIGNAL` is the way to "return" an error. `SIGNAL` provides error information to a handler, to an outer portion of the application, or to the client. Also, it provides control over the error's characteristics (error number, `SQLSTATE` value, message). Without `SIGNAL`, it is necessary to resort to workarounds such as deliberately referring to a nonexistent table to cause a routine to return an error.

No privileges are required to execute the `SIGNAL` statement.

To retrieve information from the diagnostics area, use the `GET DIAGNOSTICS` statement (see Section 13.6.7.3, "GET DIAGNOSTICS Statement"). For information about the diagnostics area, see Section 13.6.7.7, "The MySQL Diagnostics Area".

- SIGNAL Overview
- Signal Condition Information Items
- Effect of Signals on Handlers, Cursors, and Statements

SIGNAL Overview

The *`condition_value`* in a `SIGNAL` statement indicates the error value to be returned. It can be an `SQLSTATE` value (a 5-character string literal) or a *`condition_name`* that refers to a named condition previously defined with `DECLARE ... CONDITION` (see Section 13.6.7.1, "DECLARE ... CONDITION Statement").

An `SQLSTATE` value can indicate errors, warnings, or "not found." The first two characters of the value indicate its error class, as discussed in Signal Condition Information Items. Some signal values cause statement termination; see Effect of Signals on Handlers, Cursors, and Statements.

The `SQLSTATE` value for a `SIGNAL` statement should not start with `'00'` because such values indicate success and are not valid for signaling an error. This is true whether the `SQLSTATE` value is specified directly in the `SIGNAL` statement or in a named condition referred to in the statement. If the value is invalid, a `Bad SQLSTATE` error occurs.

To signal a generic `SQLSTATE` value, use `'45000'`, which means "unhandled user-defined exception."

The SIGNAL statement optionally includes a SET clause that contains multiple signal items, in a list of *condition_information_item_name* = *simple_value_specification* assignments, separated by commas.

Each *condition_information_item_name* may be specified only once in the SET clause. Otherwise, a Duplicate condition information item error occurs.

Valid *simple_value_specification* designators can be specified using stored procedure or function parameters, stored program local variables declared with DECLARE, user-defined variables, system variables, or literals. A character literal may include a *_charset* introducer.

For information about permissible *condition_information_item_name* values, see Signal Condition Information Items.

The following procedure signals an error or warning depending on the value of pval, its input parameter:

```
CREATE PROCEDURE p (pval INT)
BEGIN
  DECLARE specialty CONDITION FOR SQLSTATE '45000';
  IF pval = 0 THEN
    SIGNAL SQLSTATE '01000';
  ELSEIF pval = 1 THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'An error occurred';
  ELSEIF pval = 2 THEN
    SIGNAL specialty
      SET MESSAGE_TEXT = 'An error occurred';
  ELSE
    SIGNAL SQLSTATE '01000'
      SET MESSAGE_TEXT = 'A warning occurred', MYSQL_ERRNO = 1000;
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'An error occurred', MYSQL_ERRNO = 1001;
  END IF;
END;
```

If pval is 0, p() signals a warning because SQLSTATE values that begin with '01' are signals in the warning class. The warning does not terminate the procedure, and can be seen with SHOW WARNINGS after the procedure returns.

If pval is 1, p() signals an error and sets the MESSAGE_TEXT condition information item. The error terminates the procedure, and the text is returned with the error information.

If pval is 2, the same error is signaled, although the SQLSTATE value is specified using a named condition in this case.

If pval is anything else, p() first signals a warning and sets the message text and error number condition information items. This warning does not terminate the procedure, so execution continues and p() then signals an error. The error does terminate the procedure. The message text and error number set by the warning are replaced by the values set by the error, which are returned with the error information.

SIGNAL is typically used within stored programs, but it is a MySQL extension that it is permitted outside handler context. For example, if you invoke the mysql client program, you can enter any of these statements at the prompt:

```
SIGNAL SQLSTATE '77777';

CREATE TRIGGER t_bi BEFORE INSERT ON t
  FOR EACH ROW SIGNAL SQLSTATE '77777';

CREATE EVENT e ON SCHEDULE EVERY 1 SECOND
  DO SIGNAL SQLSTATE '77777';
```

SIGNAL executes according to the following rules:

If the SIGNAL statement indicates a particular SQLSTATE value, that value is used to signal the condition specified. Example:

```
CREATE PROCEDURE p (divisor INT)
BEGIN
  IF divisor = 0 THEN
    SIGNAL SQLSTATE '22012';
  END IF;
END;
```

If the SIGNAL statement uses a named condition, the condition must be declared in some scope that applies to the SIGNAL statement, and must be defined using an SQLSTATE value, not a MySQL error number. Example:

```
CREATE PROCEDURE p (divisor INT)
BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
  IF divisor = 0 THEN
    SIGNAL divide_by_zero;
  END IF;
END;
```

If the named condition does not exist in the scope of the SIGNAL statement, an Undefined CONDITION error occurs.

If SIGNAL refers to a named condition that is defined with a MySQL error number rather than an SQLSTATE value, a SIGNAL/RESIGNAL can only use a CONDITION defined with SQLSTATE error occurs. The following statements cause that error because the named condition is associated with a MySQL error number:

```
DECLARE no_such_table CONDITION FOR 1051;
SIGNAL no_such_table;
```

If a condition with a given name is declared multiple times in different scopes, the declaration with the most local scope applies. Consider the following procedure:

```
CREATE PROCEDURE p (divisor INT)
BEGIN
  DECLARE my_error CONDITION FOR SQLSTATE '45000';
  IF divisor = 0 THEN
    BEGIN
      DECLARE my_error CONDITION FOR SQLSTATE '22012';
      SIGNAL my_error;
    END;
  END IF;
  SIGNAL my_error;
END;
```

If divisor is 0, the first SIGNAL statement executes. The innermost my_error condition declaration applies, raising SQLSTATE '22012'.

If divisor is not 0, the second SIGNAL statement executes. The outermost my_error condition declaration applies, raising SQLSTATE '45000'.

For information about how the server chooses handlers when a condition occurs, see Section 13.6.7.6, “Scope Rules for Handlers”.

Signals can be raised within exception handlers:

```
CREATE PROCEDURE p ()
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    SIGNAL SQLSTATE VALUE '99999'
      SET MESSAGE_TEXT = 'An error occurred';
  END;
  DROP TABLE no_such_table;
END;
```

`CALL p()` reaches the `DROP TABLE` statement. There is no table named `no_such_table`, so the error handler is activated. The error handler destroys the original error ("no such table") and makes a new error with `SQLSTATE '99999'` and message `An error occurred`.

Signal Condition Information Items

The following table lists the names of diagnostics area condition information items that can be set in a `SIGNAL` (or `RESIGNAL`) statement. All items are standard SQL except `MYSQL_ERRNO`, which is a MySQL extension. For more information about these items see Section 13.6.7.7, "The MySQL Diagnostics Area".

```
Item Name             Definition
---------             ----------
CLASS_ORIGIN          VARCHAR(64)
SUBCLASS_ORIGIN       VARCHAR(64)
CONSTRAINT_CATALOG    VARCHAR(64)
CONSTRAINT_SCHEMA     VARCHAR(64)
CONSTRAINT_NAME       VARCHAR(64)
CATALOG_NAME          VARCHAR(64)
SCHEMA_NAME           VARCHAR(64)
TABLE_NAME            VARCHAR(64)
COLUMN_NAME           VARCHAR(64)
CURSOR_NAME           VARCHAR(64)
MESSAGE_TEXT          VARCHAR(128)
MYSQL_ERRNO           SMALLINT UNSIGNED
```

The character set for character items is UTF-8.

It is illegal to assign `NULL` to a condition information item in a `SIGNAL` statement.

A `SIGNAL` statement always specifies an `SQLSTATE` value, either directly, or indirectly by referring to a named condition defined with an `SQLSTATE` value. The first two characters of an `SQLSTATE` value are its class, and the class determines the default value for the condition information items:

- Class = `'00'` (success)

  Illegal. `SQLSTATE` values that begin with `'00'` indicate success and are not valid for `SIGNAL`.

- Class = `'01'` (warning)

  ```
  MESSAGE_TEXT = 'Unhandled user-defined warning condition';
  MYSQL_ERRNO = ER_SIGNAL_WARN
  ```

- Class = `'02'` (not found)

  ```
  MESSAGE_TEXT = 'Unhandled user-defined not found condition';
  MYSQL_ERRNO = ER_SIGNAL_NOT_FOUND
  ```

- Class > `'02'` (exception)

  ```
  MESSAGE_TEXT = 'Unhandled user-defined exception condition';
  MYSQL_ERRNO = ER_SIGNAL_EXCEPTION
  ```

For legal classes, the other condition information items are set as follows:

```
CLASS_ORIGIN = SUBCLASS_ORIGIN = '';
CONSTRAINT_CATALOG = CONSTRAINT_SCHEMA = CONSTRAINT_NAME = '';
CATALOG_NAME = SCHEMA_NAME = TABLE_NAME = COLUMN_NAME = '';
CURSOR_NAME = '';
```

The error values that are accessible after `SIGNAL` executes are the `SQLSTATE` value raised by the `SIGNAL` statement and the `MESSAGE_TEXT` and `MYSQL_ERRNO` items. These values are available from the C API:

- `mysql_sqlstate()` returns the `SQLSTATE` value.
- `mysql_errno()` returns the `MYSQL_ERRNO` value.

- `mysql_error()` returns the `MESSAGE_TEXT` value.

At the SQL level, the output from `SHOW WARNINGS` and `SHOW ERRORS` indicates the `MYSQL_ERRNO` and `MESSAGE_TEXT` values in the `Code` and `Message` columns.

To retrieve information from the diagnostics area, use the `GET DIAGNOSTICS` statement (see Section 13.6.7.3, “GET DIAGNOSTICS Statement”). For information about the diagnostics area, see Section 13.6.7.7, “The MySQL Diagnostics Area”.

Effect of Signals on Handlers, Cursors, and Statements

Signals have different effects on statement execution depending on the signal class. The class determines how severe an error is. MySQL ignores the value of the `sql_mode` system variable; in particular, strict SQL mode does not matter. MySQL also ignores `IGNORE`: The intent of `SIGNAL` is to raise a user-generated error explicitly, so a signal is never ignored.

In the following descriptions, “unhandled” means that no handler for the signaled `SQLSTATE` value has been defined with `DECLARE ... HANDLER`.

- Class = `'00'` (success)

  Illegal. `SQLSTATE` values that begin with `'00'` indicate success and are not valid for `SIGNAL`.

- Class = `'01'` (warning)

  The value of the `warning_count` system variable goes up. `SHOW WARNINGS` shows the signal. `SQLWARNING` handlers catch the signal.

  Warnings cannot be returned from stored functions because the `RETURN` statement that causes the function to return clears the diagnostic area. The statement thus clears any warnings that may have been present there (and resets `warning_count` to 0).

- Class = `'02'` (not found)

  `NOT FOUND` handlers catch the signal. There is no effect on cursors. If the signal is unhandled in a stored function, statements end.

- Class > `'02'` (exception)

  `SQLEXCEPTION` handlers catch the signal. If the signal is unhandled in a stored function, statements end.

- Class = `'40'`

  Treated as an ordinary exception.

13.6.7.6 Scope Rules for Handlers

A stored program may include handlers to be invoked when certain conditions occur within the program. The applicability of each handler depends on its location within the program definition and on the condition or conditions that it handles:

- A handler declared in a `BEGIN ... END` block is in scope only for the SQL statements following the handler declarations in the block. If the handler itself raises a condition, it cannot handle that condition, nor can any other handlers declared in the block. In the following example, handlers `H1` and `H2` are in scope for conditions raised by statements *`stmt1`* and *`stmt2`*. But neither `H1` nor `H2` are in scope for conditions raised in the body of `H1` or `H2`.

```
BEGIN -- outer block
  DECLARE EXIT HANDLER FOR ...;  -- handler H1
  DECLARE EXIT HANDLER FOR ...;  -- handler H2
  stmt1;
  stmt2;
END;
```

- A handler is in scope only for the block in which it is declared, and cannot be activated for conditions occurring outside that block. In the following example, handler `H1` is in scope for *`stmt1`* in the inner block, but not for *`stmt2`* in the outer block:

```
BEGIN -- outer block
  BEGIN -- inner block
    DECLARE EXIT HANDLER FOR ...;  -- handler H1
    stmt1;
  END;
  stmt2;
END;
```

- A handler can be specific or general. A specific handler is for a MySQL error code, `SQLSTATE` value, or condition name. A general handler is for a condition in the `SQLWARNING`, `SQLEXCEPTION`, or `NOT FOUND` class. Condition specificity is related to condition precedence, as described later.

Multiple handlers can be declared in different scopes and with different specificities. For example, there might be a specific MySQL error code handler in an outer block, and a general `SQLWARNING` handler in an inner block. Or there might be handlers for a specific MySQL error code and the general `SQLWARNING` class in the same block.

Whether a handler is activated depends not only on its own scope and condition value, but on what other handlers are present. When a condition occurs in a stored program, the server searches for applicable handlers in the current scope (current `BEGIN ... END` block). If there are no applicable handlers, the search continues outward with the handlers in each successive containing scope (block). When the server finds one or more applicable handlers at a given scope, it chooses among them based on condition precedence:

- A MySQL error code handler takes precedence over an `SQLSTATE` value handler.
- An `SQLSTATE` value handler takes precedence over general `SQLWARNING`, `SQLEXCEPTION`, or `NOT FOUND` handlers.
- An `SQLEXCEPTION` handler takes precedence over an `SQLWARNING` handler.
- It is possible to have several applicable handlers with the same precedence. For example, a statement could generate multiple warnings with different error codes, for each of which an error-specific handler exists. In this case, the choice of which handler the server activates is nondeterministic, and may change depending on the circumstances under which the condition occurs.

One implication of the handler selection rules is that if multiple applicable handlers occur in different scopes, handlers with the most local scope take precedence over handlers in outer scopes, even over those for more specific conditions.

If there is no appropriate handler when a condition occurs, the action taken depends on the class of the condition:

- For `SQLEXCEPTION` conditions, the stored program terminates at the statement that raised the condition, as if there were an `EXIT` handler. If the program was called by another stored program, the calling program handles the condition using the handler selection rules applied to its own handlers.
- For `SQLWARNING` conditions, the program continues executing, as if there were a `CONTINUE` handler.
- For `NOT FOUND` conditions, if the condition was raised normally, the action is `CONTINUE`. If it was raised by `SIGNAL` or `RESIGNAL`, the action is `EXIT`.

The following examples demonstrate how MySQL applies the handler selection rules.

This procedure contains two handlers, one for the specific `SQLSTATE` value (`'42S02'`) that occurs for attempts to drop a nonexistent table, and one for the general `SQLEXCEPTION` class:

```
CREATE PROCEDURE p1()
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'
    SELECT 'SQLSTATE handler was activated' AS msg;
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    SELECT 'SQLEXCEPTION handler was activated' AS msg;

  DROP TABLE test.t;
END;
```

Both handlers are declared in the same block and have the same scope. However, `SQLSTATE` handlers take precedence over `SQLEXCEPTION` handlers, so if the table `t` is nonexistent, the `DROP TABLE` statement raises a condition that activates the `SQLSTATE` handler:

```
mysql> CALL p1();
+-------------------------------+
| msg                           |
+-------------------------------+
| SQLSTATE handler was activated |
+-------------------------------+
```

This procedure contains the same two handlers. But this time, the `DROP TABLE` statement and `SQLEXCEPTION` handler are in an inner block relative to the `SQLSTATE` handler:

```
CREATE PROCEDURE p2()
BEGIN -- outer block
    DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'
      SELECT 'SQLSTATE handler was activated' AS msg;
  BEGIN -- inner block
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
      SELECT 'SQLEXCEPTION handler was activated' AS msg;

    DROP TABLE test.t; -- occurs within inner block
  END;
END;
```

In this case, the handler that is more local to where the condition occurs takes precedence. The `SQLEXCEPTION` handler activates, even though it is more general than the `SQLSTATE` handler:

```
mysql> CALL p2();
+-----------------------------------+
| msg                               |
+-----------------------------------+
| SQLEXCEPTION handler was activated |
+-----------------------------------+
```

In this procedure, one of the handlers is declared in a block inner to the scope of the `DROP TABLE` statement:

```
CREATE PROCEDURE p3()
BEGIN -- outer block
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    SELECT 'SQLEXCEPTION handler was activated' AS msg;
  BEGIN -- inner block
    DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'
      SELECT 'SQLSTATE handler was activated' AS msg;
  END;

  DROP TABLE test.t; -- occurs within outer block
END;
```

Only the `SQLEXCEPTION` handler applies because the other one is not in scope for the condition raised by the `DROP TABLE`:

```
mysql> CALL p3();
+-----------------------------------+
| msg                               |
+-----------------------------------+
| SQLEXCEPTION handler was activated |
+-----------------------------------+
```

In this procedure, both handlers are declared in a block inner to the scope of the `DROP TABLE` statement:

```
CREATE PROCEDURE p4()
BEGIN -- outer block
  BEGIN -- inner block
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
      SELECT 'SQLEXCEPTION handler was activated' AS msg;
    DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'
      SELECT 'SQLSTATE handler was activated' AS msg;
  END;

  DROP TABLE test.t; -- occurs within outer block
END;
```

Neither handler applies because they are not in scope for the `DROP TABLE`. The condition raised by the statement goes unhandled and terminates the procedure with an error:

```
mysql> CALL p4();
ERROR 1051 (42S02): Unknown table 'test.t'
```

13.6.7.7 The MySQL Diagnostics Area

SQL statements produce diagnostic information that populates the diagnostics area. Standard SQL has a diagnostics area stack, containing a diagnostics area for each nested execution context. Standard SQL also supports `GET STACKED DIAGNOSTICS` syntax for referring to the second diagnostics area during condition handler execution.

The following discussion describes the structure of the diagnostics area in MySQL, the information items recognized by MySQL, how statements clear and set the diagnostics area, and how diagnostics areas are pushed to and popped from the stack.

- Diagnostics Area Structure
- Diagnostics Area Information Items
- How the Diagnostics Area is Cleared and Populated
- How the Diagnostics Area Stack Works
- Diagnostics Area-Related System Variables

Diagnostics Area Structure

The diagnostics area contains two kinds of information:

- Statement information, such as the number of conditions that occurred or the affected-rows count.
- Condition information, such as the error code and message. If a statement raises multiple conditions, this part of the diagnostics area has a condition area for each one. If a statement raises no conditions, this part of the diagnostics area is empty.

For a statement that produces three conditions, the diagnostics area contains statement and condition information like this:

```
Statement information:
  row count
  ... other statement information items ...
Condition area list:
  Condition area 1:
    error code for condition 1
    error message for condition 1
    ... other condition information items ...
  Condition area 2:
    error code for condition 2:
    error message for condition 2
```

```
    ... other condition information items ...
  Condition area 3:
    error code for condition 3
    error message for condition 3
    ... other condition information items ...
```

Diagnostics Area Information Items

The diagnostics area contains statement and condition information items. Numeric items are integers. The character set for character items is UTF-8. No item can be `NULL`. If a statement or condition item is not set by a statement that populates the diagnostics area, its value is 0 or the empty string, depending on the item data type.

The statement information part of the diagnostics area contains these items:

- `NUMBER`: An integer indicating the number of condition areas that have information.
- `ROW_COUNT`: An integer indicating the number of rows affected by the statement. `ROW_COUNT` has the same value as the `ROW_COUNT()` function (see Section 12.16, “Information Functions”).

The condition information part of the diagnostics area contains a condition area for each condition. Condition areas are numbered from 1 to the value of the `NUMBER` statement condition item. If `NUMBER` is 0, there are no condition areas.

Each condition area contains the items in the following list. All items are standard SQL except `MYSQL_ERRNO`, which is a MySQL extension. The definitions apply for conditions generated other than by a signal (that is, by a `SIGNAL` or `RESIGNAL` statement). For nonsignal conditions, MySQL populates only those condition items not described as always empty. The effects of signals on the condition area are described later.

- `CLASS_ORIGIN`: A string containing the class of the `RETURNED_SQLSTATE` value. If the `RETURNED_SQLSTATE` value begins with a class value defined in SQL standards document ISO 9075-2 (section 24.1, SQLSTATE), `CLASS_ORIGIN` is `'ISO 9075'`. Otherwise, `CLASS_ORIGIN` is `'MySQL'`.
- `SUBCLASS_ORIGIN`: A string containing the subclass of the `RETURNED_SQLSTATE` value. If `CLASS_ORIGIN` is `'ISO 9075'` or `RETURNED_SQLSTATE` ends with `'000'`, `SUBCLASS_ORIGIN` is `'ISO 9075'`. Otherwise, `SUBCLASS_ORIGIN` is `'MySQL'`.
- `RETURNED_SQLSTATE`: A string that indicates the `SQLSTATE` value for the condition.
- `MESSAGE_TEXT`: A string that indicates the error message for the condition.
- `MYSQL_ERRNO`: An integer that indicates the MySQL error code for the condition.
- `CONSTRAINT_CATALOG`, `CONSTRAINT_SCHEMA`, `CONSTRAINT_NAME`: Strings that indicate the catalog, schema, and name for a violated constraint. They are always empty.
- `CATALOG_NAME`, `SCHEMA_NAME`, `TABLE_NAME`, `COLUMN_NAME`: Strings that indicate the catalog, schema, table, and column related to the condition. They are always empty.
- `CURSOR_NAME`: A string that indicates the cursor name. This is always empty.

For the `RETURNED_SQLSTATE`, `MESSAGE_TEXT`, and `MYSQL_ERRNO` values for particular errors, see Server Error Message Reference.

If a `SIGNAL` (or `RESIGNAL`) statement populates the diagnostics area, its `SET` clause can assign to any condition information item except `RETURNED_SQLSTATE` any value that is legal for the item data type. `SIGNAL` also sets the `RETURNED_SQLSTATE` value, but not directly in its `SET` clause. That value comes from the `SIGNAL` statement `SQLSTATE` argument.

`SIGNAL` also sets statement information items. It sets `NUMBER` to 1. It sets `ROW_COUNT` to −1 for errors and 0 otherwise.

How the Diagnostics Area is Cleared and Populated

Nondiagnostic SQL statements populate the diagnostics area automatically, and its contents can be set explicitly with the `SIGNAL` and `RESIGNAL` statements. The diagnostics area can be examined with `GET DIAGNOSTICS` to extract specific items, or with `SHOW WARNINGS` or `SHOW ERRORS` to see conditions or errors.

SQL statements clear and set the diagnostics area as follows:

- When the server starts executing a statement after parsing it, it clears the diagnostics area for nondiagnostic statements. Diagnostic statements do not clear the diagnostics area. These statements are diagnostic:
  - `GET DIAGNOSTICS`
  - `SHOW ERRORS`
  - `SHOW WARNINGS`
- If a statement raises a condition, the diagnostics area is cleared of conditions that belong to earlier statements. The exception is that conditions raised by `GET DIAGNOSTICS` and `RESIGNAL` are added to the diagnostics area without clearing it.

Thus, even a statement that does not normally clear the diagnostics area when it begins executing clears it if the statement raises a condition.

The following example shows the effect of various statements on the diagnostics area, using `SHOW WARNINGS` to display information about conditions stored there.

This `DROP TABLE` statement clears the diagnostics area and populates it when the condition occurs:

```
mysql> DROP TABLE IF EXISTS test.no_such_table;
Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> SHOW WARNINGS;
+-------+------+-----------------------------------+
| Level | Code | Message                           |
+-------+------+-----------------------------------+
| Note  | 1051 | Unknown table 'test.no_such_table' |
+-------+------+-----------------------------------+
1 row in set (0.00 sec)
```

This `SET` statement generates an error, so it clears and populates the diagnostics area:

```
mysql> SET @x = @@x;
ERROR 1193 (HY000): Unknown system variable 'x'

mysql> SHOW WARNINGS;
+-------+------+----------------------------+
| Level | Code | Message                    |
+-------+------+----------------------------+
| Error | 1193 | Unknown system variable 'x' |
+-------+------+----------------------------+
1 row in set (0.00 sec)
```

The previous `SET` statement produced a single condition, so 1 is the only valid condition number for `GET DIAGNOSTICS` at this point. The following statement uses a condition number of 2, which produces a warning that is added to the diagnostics area without clearing it:

```
mysql> GET DIAGNOSTICS CONDITION 2 @p = MESSAGE_TEXT;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+-------+------+----------------------------+
| Level | Code | Message                    |
+-------+------+----------------------------+
| Error | 1193 | Unknown system variable 'xx' |
```

```
| Error | 1753 | Invalid condition number     |
+-------+------+------------------------------+
2 rows in set (0.00 sec)
```

Now there are two conditions in the diagnostics area, so the same `GET DIAGNOSTICS` statement succeeds:

```
mysql> GET DIAGNOSTICS CONDITION 2 @p = MESSAGE_TEXT;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @p;
+--------------------------+
| @p                       |
+--------------------------+
| Invalid condition number |
+--------------------------+
1 row in set (0.01 sec)
```

How the Diagnostics Area Stack Works

When a push to the diagnostics area stack occurs, the first (current) diagnostics area becomes the second (stacked) diagnostics area and a new current diagnostics area is created as a copy of it. Diagnostics areas are pushed to and popped from the stack under the following circumstances:

- Execution of a stored program

  A push occurs before the program executes and a pop occurs afterward. If the stored program ends while handlers are executing, there can be more than one diagnostics area to pop; this occurs due to an exception for which there are no appropriate handlers or due to `RETURN` in the handler.

  Any warning or error conditions in the popped diagnostics areas then are added to the current diagnostics area, except that, for triggers, only errors are added. When the stored program ends, the caller sees these conditions in its current diagnostics area.

- Execution of a condition handler within a stored program

  When a push occurs as a result of condition handler activation, the stacked diagnostics area is the area that was current within the stored program prior to the push. The new now-current diagnostics area is the handler's current diagnostics area. `GET [CURRENT] DIAGNOSTICS` and `GET STACKED DIAGNOSTICS` can be used within the handler to access the contents of the current (handler) and stacked (stored program) diagnostics areas. Initially, they return the same result, but statements executing within the handler modify the current diagnostics area, clearing and setting its contents according to the normal rules (see How the Diagnostics Area is Cleared and Populated). The stacked diagnostics area cannot be modified by statements executing within the handler except `RESIGNAL`.

  If the handler executes successfully, the current (handler) diagnostics area is popped and the stacked (stored program) diagnostics area again becomes the current diagnostics area. Conditions added to the handler diagnostics area during handler execution are added to the current diagnostics area.

- Execution of `RESIGNAL`

  The `RESIGNAL` statement passes on the error condition information that is available during execution of a condition handler within a compound statement inside a stored program. `RESIGNAL` may change some or all information before passing it on, modifying the diagnostics stack as described in Section 13.6.7.4, “RESIGNAL Statement”.

Diagnostics Area-Related System Variables

Certain system variables control or are related to some aspects of the diagnostics area:

- `max_error_count` controls the number of condition areas in the diagnostics area. If more conditions than this occur, MySQL silently discards information for the excess conditions. (Conditions

added by `RESIGNAL` are always added, with older conditions being discarded as necessary to make room.)

- `warning_count` indicates the number of conditions that occurred. This includes errors, warnings, and notes. Normally, `NUMBER` and `warning_count` are the same. However, as the number of conditions generated exceeds `max_error_count`, the value of `warning_count` continues to rise whereas `NUMBER` remains capped at `max_error_count` because no additional conditions are stored in the diagnostics area.
- `error_count` indicates the number of errors that occurred. This value includes “not found” and exception conditions, but excludes warnings and notes. Like `warning_count`, its value can exceed `max_error_count`.
- If the `sql_notes` system variable is set to 0, notes are not stored and do not increment `warning_count`.

Example: If `max_error_count` is 10, the diagnostics area can contain a maximum of 10 condition areas. Suppose that a statement raises 20 conditions, 12 of which are errors. In that case, the diagnostics area contains the first 10 conditions, `NUMBER` is 10, `warning_count` is 20, and `error_count` is 12.

Changes to the value of `max_error_count` have no effect until the next attempt to modify the diagnostics area. If the diagnostics area contains 10 condition areas and `max_error_count` is set to 5, that has no immediate effect on the size or content of the diagnostics area.

13.6.7.8 Condition Handling and OUT or INOUT Parameters

If a stored procedure exits with an unhandled exception, modified values of `OUT` and `INOUT` parameters are not propagated back to the caller.

If an exception is handled by a `CONTINUE` or `EXIT` handler that contains a `RESIGNAL` statement, execution of `RESIGNAL` pops the Diagnostics Area stack, thus signalling the exception (that is, the information that existed before entry into the handler). If the exception is an error, the values of `OUT` and `INOUT` parameters are not propagated back to the caller.

13.6.8 Restrictions on Condition Handling

`SIGNAL`, `RESIGNAL`, and `GET DIAGNOSTICS` are not permissible as prepared statements. For example, this statement is invalid:

```
PREPARE stmt1 FROM 'SIGNAL SQLSTATE "02000"';
```

`SQLSTATE` values in class `'04'` are not treated specially. They are handled the same as other exceptions.

In standard SQL, the first condition relates to the `SQLSTATE` value returned for the previous SQL statement. In MySQL, this is not guaranteed, so to get the main error, you cannot do this:

```
GET DIAGNOSTICS CONDITION 1 @errno = MYSQL_ERRNO;
```

Instead, do this:

```
GET DIAGNOSTICS @cno = NUMBER;
GET DIAGNOSTICS CONDITION @cno @errno = MYSQL_ERRNO;
```

13.7 Database Administration Statements

13.7.1 Account Management Statements

MySQL account information is stored in the tables of the `mysql` system schema. This database and the access control system are discussed extensively in Chapter 5, *MySQL Server Administration*, which you should consult for additional details.

Important

Some MySQL releases introduce changes to the grant tables to add new privileges or features. To make sure that you can take advantage of any new capabilities, update your grant tables to the current structure whenever you upgrade MySQL. See Section 2.10, “Upgrading MySQL”.

When the `read_only` system variable is enabled, account-management statements require the `CONNECTION_ADMIN` privilege (or the deprecated `SUPER` privilege), in addition to any other required privileges. This is because they modify tables in the `mysql` system schema.

Account management statements are atomic and crash safe. For more information, see Section 13.1.1, “Atomic Data Definition Statement Support”.

13.7.1.1 ALTER USER Statement

```
ALTER USER [IF EXISTS]
    user [auth_option] [, user [auth_option]] ...
    [REQUIRE {NONE | tls_option [[AND] tls_option] ...}]
    [WITH resource_option [resource_option] ...]
    [password_option | lock_option] ...
    [COMMENT 'comment_string' | ATTRIBUTE 'json_object']

ALTER USER [IF EXISTS]
    USER() user_func_auth_option

ALTER USER [IF EXISTS]
    user [registration_option]

ALTER USER [IF EXISTS]
    USER() [registration_option]

ALTER USER [IF EXISTS]
    user DEFAULT ROLE
    {NONE | ALL | role [, role ] ...}

user:
    (see Section 6.2.4, “Specifying Account Names”)

auth_option: {
    IDENTIFIED BY 'auth_string'
        [REPLACE 'current_auth_string']
        [RETAIN CURRENT PASSWORD]
  | IDENTIFIED BY RANDOM PASSWORD
        [REPLACE 'current_auth_string']
        [RETAIN CURRENT PASSWORD]
  | IDENTIFIED WITH auth_plugin
  | IDENTIFIED WITH auth_plugin BY 'auth_string'
        [REPLACE 'current_auth_string']
        [RETAIN CURRENT PASSWORD]
  | IDENTIFIED WITH auth_plugin BY RANDOM PASSWORD
        [REPLACE 'current_auth_string']
        [RETAIN CURRENT PASSWORD]
  | IDENTIFIED WITH auth_plugin AS 'auth_string'
  | DISCARD OLD PASSWORD
  | ADD factor factor_auth_option [ADD factor factor_auth_option]
  | MODIFY factor factor_auth_option [MODIFY factor factor_auth_option]
  | DROP factor [DROP factor]
}

user_func_auth_option: {
    IDENTIFIED BY 'auth_string'
        [REPLACE 'current_auth_string']
        [RETAIN CURRENT PASSWORD]
  | DISCARD OLD PASSWORD
}

factor_auth_option: {
    IDENTIFIED BY 'auth_string'
```

```
  | IDENTIFIED BY RANDOM PASSWORD
  | IDENTIFIED WITH auth_plugin BY 'auth_string'
  | IDENTIFIED WITH auth_plugin BY RANDOM PASSWORD
  | IDENTIFIED WITH auth_plugin AS 'auth_string'
}

registration_option: {
    factor INITIATE REGISTRATION
  | factor FINISH REGISTRATION SET CHALLENGE_RESPONSE AS 'auth_string'
  | factor UNREGISTER
}

factor: {2 | 3} FACTOR

tls_option: {
   SSL
 | X509
 | CIPHER 'cipher'
 | ISSUER 'issuer'
 | SUBJECT 'subject'
}

resource_option: {
    MAX_QUERIES_PER_HOUR count
  | MAX_UPDATES_PER_HOUR count
  | MAX_CONNECTIONS_PER_HOUR count
  | MAX_USER_CONNECTIONS count
}

password_option: {
    PASSWORD EXPIRE [DEFAULT | NEVER | INTERVAL N DAY]
  | PASSWORD HISTORY {DEFAULT | N}
  | PASSWORD REUSE INTERVAL {DEFAULT | N DAY}
  | PASSWORD REQUIRE CURRENT [DEFAULT | OPTIONAL]
  | FAILED_LOGIN_ATTEMPTS N
  | PASSWORD_LOCK_TIME {N | UNBOUNDED}
}

lock_option: {
    ACCOUNT LOCK
  | ACCOUNT UNLOCK
}
```

The `ALTER USER` statement modifies MySQL accounts. It enables authentication, role, SSL/TLS, resource-limit, password-management, comment, and attribute properties to be modified for existing accounts. It can also be used to lock and unlock accounts.

In most cases, `ALTER USER` requires the global `CREATE USER` privilege, or the `UPDATE` privilege for the `mysql` system schema. The exceptions are:

- Any client who connects to the server using a nonanonymous account can change the password for that account. (In particular, you can change your own password.) To see which account the server authenticated you as, invoke the `CURRENT_USER()` function:

  ```
  SELECT CURRENT_USER();
  ```

- For `DEFAULT ROLE` syntax, `ALTER USER` requires these privileges:

  - Setting the default roles for another user requires the global `CREATE USER` privilege, or the `UPDATE` privilege for the `mysql.default_roles` system table.

  - Setting the default roles for yourself requires no special privileges, as long as the roles you want as the default have been granted to you.

- Statements that modify secondary passwords require these privileges:

  - The `APPLICATION_PASSWORD_ADMIN` privilege is required to use the `RETAIN CURRENT PASSWORD` or `DISCARD OLD PASSWORD` clause for `ALTER USER` statements that apply to your

own account. The privilege is required to manipulate your own secondary password because most users require only one password.

- If an account is to be permitted to manipulate secondary passwords for all accounts, it requires the `CREATE USER` privilege rather than `APPLICATION_PASSWORD_ADMIN`.

When the `read_only` system variable is enabled, `ALTER USER` additionally requires the `CONNECTION_ADMIN` privilege (or the deprecated `SUPER` privilege).

As of MySQL 8.0.27, these additional privilege considerations apply:

- The `authentication_policy` system variable places certain constraints on how the authentication-related clauses of `ALTER USER` statements may be used; for details, see the description of that variable. These constraints do not apply if you have the `AUTHENTICATION_POLICY_ADMIN` privilege.
- To modify an account that uses passwordless authentication, you must have the `PASSWORDLESS_USER_ADMIN` privilege.

By default, an error occurs if you try to modify a user that does not exist. If the `IF EXISTS` clause is given, the statement produces a warning for each named user that does not exist, rather than an error.

Important

Under some circumstances, `ALTER USER` may be recorded in server logs or on the client side in a history file such as `~/.mysql_history`, which means that cleartext passwords may be read by anyone having read access to that information. For information about the conditions under which this occurs for the server logs and how to control it, see Section 6.1.2.3, “Passwords and Logging”. For similar information about client-side logging, see Section 4.5.1.3, “mysql Client Logging”.

There are several aspects to the `ALTER USER` statement, described under the following topics:

- ALTER USER Overview
- ALTER USER Authentication Options
- ALTER USER Multifactor Authentication Options
- ALTER USER Registration Options
- ALTER USER Role Options
- ALTER USER SSL/TLS Options
- ALTER USER Resource-Limit Options
- ALTER USER Password-Management Options
- ALTER USER Comment and Attribute Options
- ALTER USER Account-Locking Options
- ALTER USER Binary Logging

ALTER USER Overview

For each affected account, `ALTER USER` modifies the corresponding row in the `mysql.user` system table to reflect the properties specified in the statement. Unspecified properties retain their current values.

Each account name uses the format described in Section 6.2.4, “Specifying Account Names”. The host name part of the account name, if omitted, defaults to `'%'`. It is also possible to specify `CURRENT_USER` or `CURRENT_USER()` to refer to the account associated with the current session.

For one syntax only, the account may be specified with the `USER()` function:

```
ALTER USER USER() IDENTIFIED BY 'auth_string';
```

This syntax enables changing your own password without naming your account literally. (The syntax also supports the `REPLACE`, `RETAIN CURRENT PASSWORD`, and `DISCARD OLD PASSWORD` clauses described at ALTER USER Authentication Options.)

For `ALTER USER` syntax that permits an *`auth_option`* value to follow a *`user`* value, *`auth_option`* indicates how the account authenticates by specifying an account authentication plugin, credentials (for example, a password), or both. Each *`auth_option`* value applies *only* to the account named immediately preceding it.

Following the *`user`* specifications, the statement may include options for SSL/TLS, resource-limit, password-management, and locking properties. All such options are *global* to the statement and apply to *all* accounts named in the statement.

Example: Change an account's password and expire it. As a result, the user must connect with the named password and choose a new one at the next connection:

```
ALTER USER 'jeffrey'@'localhost'
  IDENTIFIED BY 'new_password' PASSWORD EXPIRE;
```

Example: Modify an account to use the `caching_sha2_password` authentication plugin and the given password. Require that a new password be chosen every 180 days, and enable failed-login tracking, such that three consecutive incorrect passwords cause temporary account locking for two days:

```
ALTER USER 'jeffrey'@'localhost'
  IDENTIFIED WITH caching_sha2_password BY 'new_password'
  PASSWORD EXPIRE INTERVAL 180 DAY
  FAILED_LOGIN_ATTEMPTS 3 PASSWORD_LOCK_TIME 2;
```

Example: Lock or unlock an account:

```
ALTER USER 'jeffrey'@'localhost' ACCOUNT LOCK;
ALTER USER 'jeffrey'@'localhost' ACCOUNT UNLOCK;
```

Example: Require an account to connect using SSL and establish a limit of 20 connections per hour:

```
ALTER USER 'jeffrey'@'localhost'
  REQUIRE SSL WITH MAX_CONNECTIONS_PER_HOUR 20;
```

Example: Alter multiple accounts, specifying some per-account properties and some global properties:

```
ALTER USER
  'jeffrey'@'localhost'
    IDENTIFIED BY 'jeffrey_new_password',
  'jeanne'@'localhost',
  'josh'@'localhost'
    IDENTIFIED BY 'josh_new_password'
    REPLACE 'josh_current_password'
    RETAIN CURRENT PASSWORD
  REQUIRE SSL WITH MAX_USER_CONNECTIONS 2
  PASSWORD HISTORY 5;
```

The `IDENTIFIED BY` value following `jeffrey` applies only to its immediately preceding account, so it changes the password to `'jeffrey_new_password'` only for `jeffrey`. For `jeanne`, there is no per-account value (thus leaving the password unchanged). For `josh`, `IDENTIFIED BY` establishes a new password (`'josh_new_password'`), `REPLACE` is specified to verify that the user issuing the `ALTER USER` statement knows the current password (`'josh_current_password'`), and that current

password is also retained as the account secondary password. (As a result, `josh` can connect with either the primary or secondary password.)

The remaining properties apply globally to all accounts named in the statement, so for both accounts:

- Connections are required to use SSL.
- The account can be used for a maximum of two simultaneous connections.
- Password changes cannot reuse any of the five most recent passwords.

Example: Discard the secondary password for `josh`, leaving the account with only its primary password:

```
ALTER USER 'josh'@'localhost' DISCARD OLD PASSWORD;
```

In the absence of a particular type of option, the account remains unchanged in that respect. For example, with no locking option, the locking state of the account is not changed.

ALTER USER Authentication Options

An account name may be followed by an *`auth_option`* authentication option that specifies the account authentication plugin, credentials, or both. It may also include a password-verification clause that specifies the account current password to be replaced, and clauses that manage whether an account has a secondary password.

Note

Clauses for random password generation, password verification, and secondary passwords apply only to accounts that use an authentication plugin that stores credentials internally to MySQL. For accounts that use a plugin that performs authentication against a credentials system that is external to MySQL, password management must be handled externally against that system as well. For more information about internal credentials storage, see Section 6.2.15, “Password Management”.

- *`auth_plugin`* names an authentication plugin. The plugin name can be a quoted string literal or an unquoted name. Plugin names are stored in the `plugin` column of the `mysql.user` system table.

  For *`auth_option`* syntax that does not specify an authentication plugin, the server assigns the default plugin, determined as described in The Default Authentication Plugin. For descriptions of each plugin, see Section 6.4.1, “Authentication Plugins”.

- Credentials that are stored internally are stored in the `mysql.user` system table. An `'`*`auth_string`*`'` value or `RANDOM PASSWORD` specifies account credentials, either as a cleartext (unencrypted) string or hashed in the format expected by the authentication plugin associated with the account, respectively:

  - For syntax that uses `BY '`*`auth_string`*`'`, the string is cleartext and is passed to the authentication plugin for possible hashing. The result returned by the plugin is stored in the `mysql.user` table. A plugin may use the value as specified, in which case no hashing occurs.
  - For syntax that uses `BY RANDOM PASSWORD`, MySQL generates a random password and as cleartext and passes it to the authentication plugin for possible hashing. The result returned by the plugin is stored in the `mysql.user` table. A plugin may use the value as specified, in which case no hashing occurs.

    Randomly generated passwords are available as of MySQL 8.0.18 and have the characteristics described in Random Password Generation.

  - For syntax that uses `AS '`*`auth_string`*`'`, the string is assumed to be already in the format the authentication plugin requires, and is stored as is in the `mysql.user` table. If a plugin requires a

hashed value, the value must be already hashed in a format appropriate for the plugin; otherwise, the value cannot be used by the plugin and correct authentication of client connections does not occur.

As of MySQL 8.0.17, a hashed string can be either a string literal or a hexadecimal value. The latter corresponds to the type of value displayed by `SHOW CREATE USER` for password hashes containing unprintable characters when the `print_identified_with_as_hex` system variable is enabled.

- If an authentication plugin performs no hashing of the authentication string, the `BY 'auth_string'` and `AS 'auth_string'` clauses have the same effect: The authentication string is stored as is in the `mysql.user` system table.

- The `REPLACE 'current_auth_string'` clause performs password verification and is available as of MySQL 8.0.13. If given:

  - `REPLACE` specifies the account current password to be replaced, as a cleartext (unencrypted) string.

  - The clause must be given if password changes for the are required to specify the current password, as verification that the user attempting to make the change actually knows the current password.

  - The clause is optional if password changes for the account may but need not specify the current password.

  - The statement fails if the clause is given but does not match the current password, even if the clause is optional.

  - `REPLACE` can be specified only when changing the account password for the current user.

  For more information about password verification by specifying the current password, see Section 6.2.15, “Password Management”.

- The `RETAIN CURRENT PASSWORD` and `DISCARD OLD PASSWORD` clauses implement dual-password capability and are available as of MySQL 8.0.14. Both are optional, but if given, have the following effects:

  - `RETAIN CURRENT PASSWORD` retains an account current password as its secondary password, replacing any existing secondary password. The new password becomes the primary password, but clients can use the account to connect to the server using either the primary or secondary password. (Exception: If the new password specified by the `ALTER USER` statement is empty, the secondary password becomes empty as well, even if `RETAIN CURRENT PASSWORD` is given.)

  - If you specify `RETAIN CURRENT PASSWORD` for an account that has an empty primary password, the statement fails.

  - If an account has a secondary password and you change its primary password without specifying `RETAIN CURRENT PASSWORD`, the secondary password remains unchanged.

  - If you change the authentication plugin assigned to the account, the secondary password is discarded. If you change the authentication plugin and also specify `RETAIN CURRENT PASSWORD`, the statement fails.

  - `DISCARD OLD PASSWORD` discards the secondary password, if one exists. The account retains only its primary password, and clients can use the account to connect to the server only with the primary password.

  For more information about use of dual passwords, see Section 6.2.15, “Password Management”.

`ALTER USER` permits these *`auth_option`* syntaxes:

- `IDENTIFIED BY 'auth_string' [REPLACE 'current_auth_string'] [RETAIN CURRENT PASSWORD]`

  Sets the account authentication plugin to the default plugin, passes the cleartext `'auth_string'` value to the plugin for possible hashing, and stores the result in the account row in the `mysql.user` system table.

  The `REPLACE` clause, if given, specifies the account current password, as described previously in this section.

  The `RETAIN CURRENT PASSWORD` clause, if given, causes the account current password to be retained as its secondary password, as described previously in this section.

- `IDENTIFIED BY RANDOM PASSWORD [REPLACE 'current_auth_string'] [RETAIN CURRENT PASSWORD]`

  Sets the account authentication plugin to the default plugin, generates a random password, passes the cleartext password value to the plugin for possible hashing, and stores the result in the account row in the `mysql.user` system table. The statement also returns the cleartext password in a result set to make it available to the user or application executing the statement. For details about the result set and characteristics of randomly generated passwords, see Random Password Generation.

  The `REPLACE` clause, if given, specifies the account current password, as described previously in this section.

  The `RETAIN CURRENT PASSWORD` clause, if given, causes the account current password to be retained as its secondary password, as described previously in this section.

- `IDENTIFIED WITH auth_plugin`

  Sets the account authentication plugin to *`auth_plugin`*, clears the credentials to the empty string (the credentials are associated with the old authentication plugin, not the new one), and stores the result in the account row in the `mysql.user` system table.

  In addition, the password is marked expired. The user must choose a new one when next connecting.

- `IDENTIFIED WITH auth_plugin BY 'auth_string' [REPLACE 'current_auth_string'] [RETAIN CURRENT PASSWORD]`

  Sets the account authentication plugin to *`auth_plugin`*, passes the cleartext `'auth_string'` value to the plugin for possible hashing, and stores the result in the account row in the `mysql.user` system table.

  The `REPLACE` clause, if given, specifies the account current password, as described previously in this section.

  The `RETAIN CURRENT PASSWORD` clause, if given, causes the account current password to be retained as its secondary password, as described previously in this section.

- `IDENTIFIED WITH auth_plugin BY RANDOM PASSWORD [REPLACE 'current_auth_string'] [RETAIN CURRENT PASSWORD]`

  Sets the account authentication plugin to *`auth_plugin`*, generates a random password, passes the cleartext password value to the plugin for possible hashing, and stores the result in the account row in the `mysql.user` system table. The statement also returns the cleartext password in a result set to make it available to the user or application executing the statement. For details about the result set and characteristics of randomly generated passwords, see Random Password Generation.

  The `REPLACE` clause, if given, specifies the account current password, as described previously in this section.

The RETAIN CURRENT PASSWORD clause, if given, causes the account current password to be retained as its secondary password, as described previously in this section.

- IDENTIFIED WITH *auth_plugin* AS '*auth_string*'

  Sets the account authentication plugin to *auth_plugin* and stores the '*auth_string*' value as is in the mysql.user account row. If the plugin requires a hashed string, the string is assumed to be already hashed in the format the plugin requires.

- DISCARD OLD PASSWORD

  Discards the account secondary password, if there is one, as described previously in this section.

Example: Specify the password as cleartext; the default plugin is used:

```
ALTER USER 'jeffrey'@'localhost'
  IDENTIFIED BY 'password';
```

Example: Specify the authentication plugin, along with a cleartext password value:

```
ALTER USER 'jeffrey'@'localhost'
  IDENTIFIED WITH mysql_native_password
             BY 'password';
```

Example: Like the preceding example, but in addition, specify the current password as a cleartext value to satisfy any account requirement that the user making the change knows that password:

```
ALTER USER 'jeffrey'@'localhost'
  IDENTIFIED WITH mysql_native_password
             BY 'password'
             REPLACE 'current_password';
```

The preceding statement fails unless the current user is jeffrey because REPLACE is permitted only for changes to the current user's password.

Example: Establish a new primary password and retain the existing password as the secondary password:

```
ALTER USER 'jeffrey'@'localhost'
  IDENTIFIED BY 'new_password'
  RETAIN CURRENT PASSWORD;
```

Example: Discard the secondary password, leaving the account with only its primary password:

```
ALTER USER 'jeffery'@'localhost' DISCARD OLD PASSWORD;
```

Example: Specify the authentication plugin, along with a hashed password value:

```
ALTER USER 'jeffrey'@'localhost'
  IDENTIFIED WITH mysql_native_password
             AS '*6C8989366EAF75BB670AD8EA7A7FC1176A95CEF4';
```

For additional information about setting passwords and authentication plugins, see Section 6.2.14, “Assigning Account Passwords”, and Section 6.2.17, “Pluggable Authentication”.

ALTER USER Multifactor Authentication Options

As of MySQL 8.0.27, ALTER USER has ADD, MODIFY, and DROP clauses that enable authentication factors to be added, modified, or dropped. In each case, the clause specifies an operation to perform on one authentication factor, and optionally an operation on another authentication factor. For each operation, the *factor* item specifies the FACTOR keyword preceded by the number 2 or 3 to indicate whether the operation applies to the second or third authentication factor. (1 is not permitted in this context. To act on the first authentication factor, use the syntax described in ALTER USER Authentication Options.)

`ALTER USER` multifactor authentication clause constraints are defined by the `authentication_policy` system variable. For example, the `authentication_policy` setting controls the number of authentication factors that accounts may have, and for each factor, which authentication methods are permitted. See Configuring the Multifactor Authentication Policy.

When `ALTER USER` adds, modifies, or drops second and third factors in a single statement, operations are executed sequentially, but if any operation in the sequence fails the entire `ALTER USER` statement fails.

For `ADD`, each named factor must not already exist or it cannot be added. For `MODIFY` and `DROP`, each named factor must exist to be modified or dropped. If a second and third factor are defined, dropping the second factor causes the third factor to take its place as the second factor.

This statement drops authentication factors 2 and 3, which has the effect of converting the account from 3FA to 1FA:

```
ALTER USER 'user' DROP 2 FACTOR 3 FACTOR;
```

For additional `ADD`, `MODIFY`, and `DROP` examples, see Getting Started with Multifactor Authentication.

For information about factor-specific rules that determine the default authentication plugin for authentication clauses that do not name a plugin, see The Default Authentication Plugin.

ALTER USER Registration Options

As of MySQL 8.0.27, `ALTER USER` has clauses that enable FIDO devices to be registered and unregistered. For more information, see Using FIDO Authentication, FIDO Device Unregistration, and the `mysql` client `--fido-register-factor` option description.

The `mysql` client `--fido-register-factor` option, used for FIDO device registration, causes the `mysql` client to generate and execute `INITIATE REGISTRATION` and `FINISH REGISTRATION` statements. These statements are not intended for manual execution.

ALTER USER Role Options

`ALTER USER ... DEFAULT ROLE` defines which roles become active when the user connects to the server and authenticates, or when the user executes the `SET ROLE DEFAULT` statement during a session.

`ALTER USER ... DEFAULT ROLE` is alternative syntax for `SET DEFAULT ROLE` (see Section 13.7.1.9, “SET DEFAULT ROLE Statement”). However, `ALTER USER` can set the default for only a single user, whereas `SET DEFAULT ROLE` can set the default for multiple users. On the other hand, you can specify `CURRENT_USER` as the user name for the `ALTER USER` statement, whereas you cannot for `SET DEFAULT ROLE`.

Each user account name uses the format described previously.

Each role name uses the format described in Section 6.2.5, “Specifying Role Names”. For example:

```
ALTER USER 'joe'@'10.0.0.1' DEFAULT ROLE administrator, developer;
```

The host name part of the role name, if omitted, defaults to `'%'`.

The clause following the `DEFAULT ROLE` keywords permits these values:

- `NONE`: Set the default to `NONE` (no roles).
- `ALL`: Set the default to all roles granted to the account.
- *`role [, role ] ...`*: Set the default to the named roles, which must exist and be granted to the account at the time `ALTER USER ... DEFAULT ROLE` is executed.

ALTER USER SSL/TLS Options

MySQL can check X.509 certificate attributes in addition to the usual authentication that is based on the user name and credentials. For background information on the use of SSL/TLS with MySQL, see Section 6.3, “Using Encrypted Connections”.

To specify SSL/TLS-related options for a MySQL account, use a `REQUIRE` clause that specifies one or more *`tls_option`* values.

Order of `REQUIRE` options does not matter, but no option can be specified twice. The `AND` keyword is optional between `REQUIRE` options.

`ALTER USER` permits these *`tls_option`* values:

- `NONE`

  Indicates that all accounts named by the statement have no SSL or X.509 requirements. Unencrypted connections are permitted if the user name and password are valid. Encrypted connections can be used, at the client's option, if the client has the proper certificate and key files.

  ```
  ALTER USER 'jeffrey'@'localhost' REQUIRE NONE;
  ```

  Clients attempt to establish a secure connection by default. For clients that have `REQUIRE NONE`, the connection attempt falls back to an unencrypted connection if a secure connection cannot be established. To require an encrypted connection, a client need specify only the `--ssl-mode=REQUIRED` option; the connection attempt fails if a secure connection cannot be established.

- `SSL`

  Tells the server to permit only encrypted connections for all accounts named by the statement.

  ```
  ALTER USER 'jeffrey'@'localhost' REQUIRE SSL;
  ```

  Clients attempt to establish a secure connection by default. For accounts that have `REQUIRE SSL`, the connection attempt fails if a secure connection cannot be established.

- `X509`

  For all accounts named by the statement, requires that clients present a valid certificate, but the exact certificate, issuer, and subject do not matter. The only requirement is that it should be possible to verify its signature with one of the CA certificates. Use of X.509 certificates always implies encryption, so the `SSL` option is unnecessary in this case.

  ```
  ALTER USER 'jeffrey'@'localhost' REQUIRE X509;
  ```

  For accounts with `REQUIRE X509`, clients must specify the `--ssl-key` and `--ssl-cert` options to connect. (It is recommended but not required that `--ssl-ca` also be specified so that the public certificate provided by the server can be verified.) This is true for `ISSUER` and `SUBJECT` as well because those `REQUIRE` options imply the requirements of `X509`.

- `ISSUER '`*`issuer`*`'`

  For all accounts named by the statement, requires that clients present a valid X.509 certificate issued by CA `'`*`issuer`*`'`. If a client presents a certificate that is valid but has a different issuer, the server rejects the connection. Use of X.509 certificates always implies encryption, so the `SSL` option is unnecessary in this case.

  ```
  ALTER USER 'jeffrey'@'localhost'
    REQUIRE ISSUER '/C=SE/ST=Stockholm/L=Stockholm/
      O=MySQL/CN=CA/emailAddress=ca@example.com';
  ```

  Because `ISSUER` implies the requirements of `X509`, clients must specify the `--ssl-key` and `--ssl-cert` options to connect. (It is recommended but not required that `--ssl-ca` also be specified so that the public certificate provided by the server can be verified.)

- `SUBJECT 'subject'`

  For all accounts named by the statement, requires that clients present a valid X.509 certificate containing the subject *`subject`*. If a client presents a certificate that is valid but has a different subject, the server rejects the connection. Use of X.509 certificates always implies encryption, so the `SSL` option is unnecessary in this case.

  ```
  ALTER USER 'jeffrey'@'localhost'
    REQUIRE SUBJECT '/C=SE/ST=Stockholm/L=Stockholm/
      O=MySQL demo client certificate/
      CN=client/emailAddress=client@example.com';
  ```

  MySQL does a simple string comparison of the *`'subject'`* value to the value in the certificate, so lettercase and component ordering must be given exactly as present in the certificate.

  Because `SUBJECT` implies the requirements of `X509`, clients must specify the `--ssl-key` and `--ssl-cert` options to connect. (It is recommended but not required that `--ssl-ca` also be specified so that the public certificate provided by the server can be verified.)

- `CIPHER 'cipher'`

  For all accounts named by the statement, requires a specific cipher method for encrypting connections. This option is needed to ensure that ciphers and key lengths of sufficient strength are used. Encryption can be weak if old algorithms using short encryption keys are used.

  ```
  ALTER USER 'jeffrey'@'localhost'
    REQUIRE CIPHER 'EDH-RSA-DES-CBC3-SHA';
  ```

The `SUBJECT`, `ISSUER`, and `CIPHER` options can be combined in the `REQUIRE` clause:

```
ALTER USER 'jeffrey'@'localhost'
  REQUIRE SUBJECT '/C=SE/ST=Stockholm/L=Stockholm/
    O=MySQL demo client certificate/
    CN=client/emailAddress=client@example.com'
  AND ISSUER '/C=SE/ST=Stockholm/L=Stockholm/
    O=MySQL/CN=CA/emailAddress=ca@example.com'
  AND CIPHER 'EDH-RSA-DES-CBC3-SHA';
```

ALTER USER Resource-Limit Options

It is possible to place limits on use of server resources by an account, as discussed in Section 6.2.21, “Setting Account Resource Limits”. To do so, use a `WITH` clause that specifies one or more *`resource_option`* values.

Order of `WITH` options does not matter, except that if a given resource limit is specified multiple times, the last instance takes precedence.

`ALTER USER` permits these *`resource_option`* values:

- `MAX_QUERIES_PER_HOUR count`, `MAX_UPDATES_PER_HOUR count`, `MAX_CONNECTIONS_PER_HOUR count`

  For all accounts named by the statement, these options restrict how many queries, updates, and connections to the server are permitted to each account during any given one-hour period. If *`count`* is `0` (the default), this means that there is no limitation for the account.

- `MAX_USER_CONNECTIONS count`

  For all accounts named by the statement, restricts the maximum number of simultaneous connections to the server by each account. A nonzero *`count`* specifies the limit for the account explicitly. If *`count`* is `0` (the default), the server determines the number of simultaneous connections for the account from the global value of the `max_user_connections` system variable. If `max_user_connections` is also zero, there is no limit for the account.

Example:

```
ALTER USER 'jeffrey'@'localhost'
  WITH MAX_QUERIES_PER_HOUR 500 MAX_UPDATES_PER_HOUR 100;
```

ALTER USER Password-Management Options

`ALTER USER` supports several *`password_option`* values for password management:

- Password expiration options: You can expire an account password manually and establish its password expiration policy. Policy options do not expire the password. Instead, they determine how the server applies automatic expiration to the account based on password age, which is assessed from the date and time of the most recent account password change.
- Password reuse options: You can restrict password reuse based on number of password changes, time elapsed, or both.
- Password verification-required options: You can indicate whether attempts to change an account password must specify the current password, as verification that the user attempting to make the change actually knows the current password.
- Incorrect-password failed-login tracking options: You can cause the server to track failed login attempts and temporarily lock accounts for which too many consecutive incorrect passwords are given. The required number of failures and the lock time are configurable.

This section describes the syntax for password-management options. For information about establishing policy for password management, see Section 6.2.15, “Password Management”.

If multiple password-management options of a given type are specified, the last one takes precedence. For example, `PASSWORD EXPIRE DEFAULT PASSWORD EXPIRE NEVER` is the same as `PASSWORD EXPIRE NEVER`.

Note

Except for the options that pertain to failed-login tracking, password-management options apply only to accounts that use an authentication plugin that stores credentials internally to MySQL. For accounts that use a plugin that performs authentication against a credentials system that is external to MySQL, password management must be handled externally against that system as well. For more information about internal credentials storage, see Section 6.2.15, “Password Management”.

A client has an expired password if the account password was expired manually or the password age is considered greater than its permitted lifetime per the automatic expiration policy. In this case, the server either disconnects the client or restricts the operations permitted to it (see Section 6.2.16, “Server Handling of Expired Passwords”). Operations performed by a restricted client result in an error until the user establishes a new account password.

Note

Although it is possible to “reset” an expired password by setting it to its current value, it is preferable, as a matter of good policy, to choose a different password. DBAs can enforce non-reuse by establishing an appropriate password-reuse policy. See Password Reuse Policy.

`ALTER USER` permits these *`password_option`* values for controlling password expiration:

- `PASSWORD EXPIRE`

  Immediately marks the password expired for all accounts named by the statement.

```
ALTER USER 'jeffrey'@'localhost' PASSWORD EXPIRE;
```

- `PASSWORD EXPIRE DEFAULT`

  Sets all accounts named by the statement so that the global expiration policy applies, as specified by the `default_password_lifetime` system variable.

```
ALTER USER 'jeffrey'@'localhost' PASSWORD EXPIRE DEFAULT;
```

- `PASSWORD EXPIRE NEVER`

  This expiration option overrides the global policy for all accounts named by the statement. For each, it disables password expiration so that the password never expires.

```
ALTER USER 'jeffrey'@'localhost' PASSWORD EXPIRE NEVER;
```

- `PASSWORD EXPIRE INTERVAL N DAY`

  This expiration option overrides the global policy for all accounts named by the statement. For each, it sets the password lifetime to *`N`* days. The following statement requires the password to be changed every 180 days:

```
ALTER USER 'jeffrey'@'localhost' PASSWORD EXPIRE INTERVAL 180 DAY;
```

`ALTER USER` permits these *`password_option`* values for controlling reuse of previous passwords based on required minimum number of password changes:

- `PASSWORD HISTORY DEFAULT`

  Sets all accounts named by the statement so that the global policy about password history length applies, to prohibit reuse of passwords before the number of changes specified by the `password_history` system variable.

```
ALTER USER 'jeffrey'@'localhost' PASSWORD HISTORY DEFAULT;
```

- `PASSWORD HISTORY N`

  This history-length option overrides the global policy for all accounts named by the statement. For each, it sets the password history length to *`N`* passwords, to prohibit reusing any of the *`N`* most recently chosen passwords. The following statement prohibits reuse of any of the previous 6 passwords:

```
ALTER USER 'jeffrey'@'localhost' PASSWORD HISTORY 6;
```

`ALTER USER` permits these *`password_option`* values for controlling reuse of previous passwords based on time elapsed:

- `PASSWORD REUSE INTERVAL DEFAULT`

  Sets all statements named by the account so that the global policy about time elapsed applies, to prohibit reuse of passwords newer than the number of days specified by the `password_reuse_interval` system variable.

```
ALTER USER 'jeffrey'@'localhost' PASSWORD REUSE INTERVAL DEFAULT;
```

- `PASSWORD REUSE INTERVAL N DAY`

  This time-elapsed option overrides the global policy for all accounts named by the statement. For each, it sets the password reuse interval to *`N`* days, to prohibit reuse of passwords newer than that many days. The following statement prohibits password reuse for 360 days:

```
ALTER USER 'jeffrey'@'localhost' PASSWORD REUSE INTERVAL 360 DAY;
```

`ALTER USER` permits these *`password_option`* values for controlling whether attempts to change an account password must specify the current password, as verification that the user attempting to make the change actually knows the current password:

- `PASSWORD REQUIRE CURRENT`

  This verification option overrides the global policy for all accounts named by the statement. For each, it requires that password changes specify the current password.

  ```
  ALTER USER 'jeffrey'@'localhost' PASSWORD REQUIRE CURRENT;
  ```

- `PASSWORD REQUIRE CURRENT OPTIONAL`

  This verification option overrides the global policy for all accounts named by the statement. For each, it does not require that password changes specify the current password. (The current password may but need not be given.)

  ```
  ALTER USER 'jeffrey'@'localhost' PASSWORD REQUIRE CURRENT OPTIONAL;
  ```

- `PASSWORD REQUIRE CURRENT DEFAULT`

  Sets all statements named by the account so that the global policy about password verification applies, as specified by the `password_require_current` system variable.

  ```
  ALTER USER 'jeffrey'@'localhost' PASSWORD REQUIRE CURRENT DEFAULT;
  ```

As of MySQL 8.0.19, `ALTER USER` permits these *`password_option`* values for controlling failed-login tracking:

- `FAILED_LOGIN_ATTEMPTS` *`N`*

  Whether to track account login attempts that specify an incorrect password. *`N`* must be a number from 0 to 32767. A value of 0 disables failed-login tracking. Values greater than 0 indicate how many consecutive password failures cause temporary account locking (if `PASSWORD_LOCK_TIME` is also nonzero).

- `PASSWORD_LOCK_TIME {`*`N`*` | UNBOUNDED}`

  How long to lock the account after too many consecutive login attempts provide an incorrect password. *`N`* must be a number from 0 to 32767, or `UNBOUNDED`. A value of 0 disables temporary account locking. Values greater than 0 indicate how long to lock the account in days. A value of `UNBOUNDED` causes the account locking duration to be unbounded; once locked, the account remains in a locked state until unlocked. For information about the conditions under which unlocking occurs, see Failed-Login Tracking and Temporary Account Locking.

For failed-login tracking and temporary locking to occur, an account's `FAILED_LOGIN_ATTEMPTS` and `PASSWORD_LOCK_TIME` options both must be nonzero. The following statement modifies an account such that it remains locked for two days after four consecutive password failures:

```
ALTER USER 'jeffrey'@'localhost'
  FAILED_LOGIN_ATTEMPTS 4 PASSWORD_LOCK_TIME 2;
```

ALTER USER Comment and Attribute Options

MySQL 8.0.21 and higher supports user comments and user attributes, as described in Section 13.7.1.3, “CREATE USER Statement”. These can be modified employing `ALTER USER` by means of the `COMMENT` and `ATTRIBUTE` options, respectively. You cannot specify both options in the same `ALTER USER` statement; attempting to do so results in a syntax error.

The user comment and user attribute are stored in the Information Schema `USER_ATTRIBUTES` table as a JSON object; the user comment is stored as the value for a `comment` key in the ATTRIBUTE column of this table, as shown later in this discussion. The `COMMENT` text can be any arbitrary quoted text, and replaces any existing user comment. The `ATTRIBUTE` value must be the valid string representation of a JSON object. This is merged with any existing user attribute as if the `JSON_MERGE_PATCH()` function had been used on the existing user attribute and the new one; for any keys that are re-used, the new value overwrites the old one, as shown here:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.USER_ATTRIBUTES
    ->     WHERE USER='bill' AND HOST='localhost';
+------+-----------+----------------+
| USER | HOST      | ATTRIBUTE      |
+------+-----------+----------------+
| bill | localhost | {"foo": "bar"} |
+------+-----------+----------------+
1 row in set (0.11 sec)

mysql> ALTER USER 'bill'@'localhost' ATTRIBUTE '{"baz": "faz", "foo": "moo"}';
Query OK, 0 rows affected (0.22 sec)

mysql> SELECT * FROM INFORMATION_SCHEMA.USER_ATTRIBUTES
    ->     WHERE USER='bill' AND HOST='localhost';
+------+-----------+------------------------------+
| USER | HOST      | ATTRIBUTE                    |
+------+-----------+------------------------------+
| bill | localhost | {"baz": "faz", "foo": "moo"} |
+------+-----------+------------------------------+
1 row in set (0.00 sec)
```

To remove a key and its value from the user attribute, set the key to JSON `null` (must be lowercase and unquoted), like this:

```
mysql> ALTER USER 'bill'@'localhost' ATTRIBUTE '{"foo": null}';
Query OK, 0 rows affected (0.08 sec)

mysql> SELECT * FROM INFORMATION_SCHEMA.USER_ATTRIBUTES
    ->     WHERE USER='bill' AND HOST='localhost';
+------+-----------+----------------+
| USER | HOST      | ATTRIBUTE      |
+------+-----------+----------------+
| bill | localhost | {"baz": "faz"} |
+------+-----------+----------------+
1 row in set (0.00 sec)
```

To set an existing user comment to an empty string, use `ALTER USER ... COMMENT ''`. This leaves an empty `comment` value in the `USER_ATTRIBUTES` table; to remove the user comment completely, use `ALTER USER ... ATTRIBUTE ...` with the value for the column key set to JSON `null` (unquoted, in lower case). This is illustrated by the following sequence of SQL statements:

```
mysql> ALTER USER 'bill'@'localhost' COMMENT 'Something about Bill';
Query OK, 0 rows affected (0.06 sec)

mysql> SELECT * FROM INFORMATION_SCHEMA.USER_ATTRIBUTES
    ->     WHERE USER='bill' AND HOST='localhost';
+------+-----------+-----------------------------------------------------+
| USER | HOST      | ATTRIBUTE                                           |
+------+-----------+-----------------------------------------------------+
| bill | localhost | {"baz": "faz", "comment": "Something about Bill"} |
+------+-----------+-----------------------------------------------------+
1 row in set (0.00 sec)

mysql> ALTER USER 'bill'@'localhost' COMMENT '';
Query OK, 0 rows affected (0.09 sec)

mysql> SELECT * FROM INFORMATION_SCHEMA.USER_ATTRIBUTES
    ->     WHERE USER='bill' AND HOST='localhost';
+------+-----------+------------------------------+
| USER | HOST      | ATTRIBUTE                    |
+------+-----------+------------------------------+
| bill | localhost | {"baz": "faz", "comment": ""} |
+------+-----------+------------------------------+
1 row in set (0.00 sec)

mysql> ALTER USER 'bill'@'localhost' ATTRIBUTE '{"comment": null}';
Query OK, 0 rows affected (0.07 sec)

mysql> SELECT * FROM INFORMATION_SCHEMA.USER_ATTRIBUTES
    ->     WHERE USER='bill' AND HOST='localhost';
+------+-----------+----------------+
```

```
| USER | HOST      | ATTRIBUTE      |
+------+-----------+----------------+
| bill | localhost | {"baz": "faz"} |
+------+-----------+----------------+
1 row in set (0.00 sec)
```

ALTER USER Account-Locking Options

MySQL supports account locking and unlocking using the `ACCOUNT LOCK` and `ACCOUNT UNLOCK` options, which specify the locking state for an account. For additional discussion, see Section 6.2.20, “Account Locking”.

If multiple account-locking options are specified, the last one takes precedence.

As of MySQL 8.0.19, `ALTER USER ... UNLOCK` unlocks any account named by the statement that is temporarily locked due to too many failed logins. See Section 6.2.15, “Password Management”.

ALTER USER Binary Logging

`ALTER USER` is written to the binary log if it succeeds, but not if it fails; in that case, rollback occurs and no changes are made. A statement written to the binary log includes all named users. If the `IF EXISTS` clause is given, this includes even users that do not exist and were not altered.

If the original statement changes the credentials for a user, the statement written to the binary log specifies the applicable authentication plugin for that user, determined as follows:

- The plugin named in the original statement, if one was specified.
- Otherwise, the plugin associated with the user account if the user exists, or the default authentication plugin if the user does not exist. (If the statement written to the binary log must specify a particular authentication plugin for a user, include it in the original statement.)

If the server adds the default authentication plugin for any users in the statement written to the binary log, it writes a warning to the error log naming those users.

If the original statement specifies the `FAILED_LOGIN_ATTEMPTS` or `PASSWORD_LOCK_TIME` option, the statement written to the binary log includes the option.

`ALTER USER` statements with clauses that support multifactor authentication (MFA) are written to the binary log with the exception of `ALTER USER user factor INITIATE REGISTRATION` statements.

- `ALTER USER user factor FINISH REGISTRATION SET CHALLENGE_RESPONSE AS 'auth_string'` statements are written to the binary log as `ALTER USER user MODIFY factor IDENTIFIED WITH authentication_fido AS fido_hash_string`;
- In a replication context, the replication user requires `PASSWORDLESS_USER_ADMIN` privilege to execute `ALTER USER ... MODIFY` operations on accounts configured for passwordless authentication using the `authentication_fido` plugin.

13.7.1.2 CREATE ROLE Statement

```
CREATE ROLE [IF NOT EXISTS] role [, role ] ...
```

`CREATE ROLE` creates one or more roles, which are named collections of privileges. To use this statement, you must have the global `CREATE ROLE` or `CREATE USER` privilege. When the `read_only` system variable is enabled, `CREATE ROLE` additionally requires the `CONNECTION_ADMIN` privilege (or the deprecated `SUPER` privilege).

A role when created is locked, has no password, and is assigned the default authentication plugin. (These role attributes can be changed later with the `ALTER USER` statement, by users who have the global `CREATE USER` privilege.)

`CREATE ROLE` either succeeds for all named roles or rolls back and has no effect if any error occurs. By default, an error occurs if you try to create a role that already exists. If the `IF NOT EXISTS` clause is given, the statement produces a warning for each named role that already exists, rather than an error.

The statement is written to the binary log if it succeeds, but not if it fails; in that case, rollback occurs and no changes are made. A statement written to the binary log includes all named roles. If the `IF NOT EXISTS` clause is given, this includes even roles that already exist and were not created.

Each role name uses the format described in Section 6.2.5, “Specifying Role Names”. For example:

```
CREATE ROLE 'admin', 'developer';
CREATE ROLE 'webapp'@'localhost';
```

The host name part of the role name, if omitted, defaults to `'%'`.

For role usage examples, see Section 6.2.10, “Using Roles”.

13.7.1.3 CREATE USER Statement

```
CREATE USER [IF NOT EXISTS]
    user [auth_option] [, user [auth_option]] ...
    DEFAULT ROLE role [, role ] ...
    [REQUIRE {NONE | tls_option [[AND] tls_option] ...}]
    [WITH resource_option [resource_option] ...]
    [password_option | lock_option] ...
    [COMMENT 'comment_string' | ATTRIBUTE 'json_object']

user:
    (see Section 6.2.4, “Specifying Account Names”)

auth_option: {
    IDENTIFIED BY 'auth_string' [AND 2fa_auth_option]
  | IDENTIFIED BY RANDOM PASSWORD [AND 2fa_auth_option]
  | IDENTIFIED WITH auth_plugin [AND 2fa_auth_option]
  | IDENTIFIED WITH auth_plugin BY 'auth_string' [AND 2fa_auth_option]
  | IDENTIFIED WITH auth_plugin BY RANDOM PASSWORD [AND 2fa_auth_option]
  | IDENTIFIED WITH auth_plugin AS 'auth_string' [AND 2fa_auth_option]
  | IDENTIFIED WITH auth_plugin [initial_auth_option]
}

2fa_auth_option: {
    IDENTIFIED BY 'auth_string' [AND 3fa_auth_option]
  | IDENTIFIED BY RANDOM PASSWORD [AND 3fa_auth_option]
  | IDENTIFIED WITH auth_plugin [AND 3fa_auth_option]
  | IDENTIFIED WITH auth_plugin BY 'auth_string' [AND 3fa_auth_option]
  | IDENTIFIED WITH auth_plugin BY RANDOM PASSWORD [AND 3fa_auth_option]
  | IDENTIFIED WITH auth_plugin AS 'auth_string' [AND 3fa_auth_option]
}

3fa_auth_option: {
    IDENTIFIED BY 'auth_string'
  | IDENTIFIED BY RANDOM PASSWORD
  | IDENTIFIED WITH auth_plugin
  | IDENTIFIED WITH auth_plugin BY 'auth_string'
  | IDENTIFIED WITH auth_plugin BY RANDOM PASSWORD
  | IDENTIFIED WITH auth_plugin AS 'auth_string'
}

initial_auth_option: {
    INITIAL AUTHENTICATION IDENTIFIED BY {RANDOM PASSWORD | 'auth_string'}
  | INITIAL AUTHENTICATION IDENTIFIED WITH auth_plugin AS 'auth_string'
}

tls_option: {
   SSL
 | X509
 | CIPHER 'cipher'
 | ISSUER 'issuer'
 | SUBJECT 'subject'
```

```
}

resource_option: {
    MAX_QUERIES_PER_HOUR count
  | MAX_UPDATES_PER_HOUR count
  | MAX_CONNECTIONS_PER_HOUR count
  | MAX_USER_CONNECTIONS count
}

password_option: {
    PASSWORD EXPIRE [DEFAULT | NEVER | INTERVAL N DAY]
  | PASSWORD HISTORY {DEFAULT | N}
  | PASSWORD REUSE INTERVAL {DEFAULT | N DAY}
  | PASSWORD REQUIRE CURRENT [DEFAULT | OPTIONAL]
  | FAILED_LOGIN_ATTEMPTS N
  | PASSWORD_LOCK_TIME {N | UNBOUNDED}
}

lock_option: {
    ACCOUNT LOCK
  | ACCOUNT UNLOCK
}
```

The `CREATE USER` statement creates new MySQL accounts. It enables authentication, role, SSL/TLS, resource-limit, password-management, comment, and attribute properties to be established for new accounts. It also controls whether accounts are initially locked or unlocked.

To use `CREATE USER`, you must have the global `CREATE USER` privilege, or the `INSERT` privilege for the `mysql` system schema. When the `read_only` system variable is enabled, `CREATE USER` additionally requires the `CONNECTION_ADMIN` privilege (or the deprecated `SUPER` privilege).

As of MySQL 8.0.27, these additional privilege considerations apply:

- The `authentication_policy` system variable places certain constraints on how the authentication-related clauses of `CREATE USER` statements may be used; for details, see the description of that variable. These constraints do not apply if you have the `AUTHENTICATION_POLICY_ADMIN` privilege.
- To create an account that uses passwordless authentication, you must have the `PASSWORDLESS_USER_ADMIN` privilege.

As of MySQL 8.0.22, `CREATE USER` fails with an error if any account to be created is named as the `DEFINER` attribute for any stored object. (That is, the statement fails if creating an account would cause the account to adopt a currently orphaned stored object.) To perform the operation anyway, you must have the `SET_USER_ID` privilege; in this case, the statement succeeds with a warning rather than failing with an error. Without `SET_USER_ID`, to perform the user-creation operation, drop the orphan objects, create the account and grant its privileges, and then re-create the dropped objects. For additional information, including how to identify which objects name a given account as the `DEFINER` attribute, see Orphan Stored Objects.

`CREATE USER` either succeeds for all named users or rolls back and has no effect if any error occurs. By default, an error occurs if you try to create a user that already exists. If the `IF NOT EXISTS` clause is given, the statement produces a warning for each named user that already exists, rather than an error.

Important

Under some circumstances, `CREATE USER` may be recorded in server logs or on the client side in a history file such as `~/.mysql_history`, which means that cleartext passwords may be read by anyone having read access to that information. For information about the conditions under which this occurs for the server logs and how to control it, see Section 6.1.2.3, “Passwords and Logging”. For similar information about client-side logging, see Section 4.5.1.3, “mysql Client Logging”.

There are several aspects to the `CREATE USER` statement, described under the following topics:

- CREATE USER Overview
- CREATE USER Authentication Options
- CREATE USER Multifactor Authentication Options
- CREATE USER Role Options
- CREATE USER SSL/TLS Options
- CREATE USER Resource-Limit Options
- CREATE USER Password-Management Options
- CREATE USER Comment and Attribute Options
- CREATE USER Account-Locking Options
- CREATE USER Binary Logging

CREATE USER Overview

For each account, `CREATE USER` creates a new row in the `mysql.user` system table. The account row reflects the properties specified in the statement. Unspecified properties are set to their default values:

- Authentication: The default authentication plugin (determined as described in The Default Authentication Plugin), and empty credentials
- Default role: `NONE`
- SSL/TLS: `NONE`
- Resource limits: Unlimited
- Password management: `PASSWORD EXPIRE DEFAULT PASSWORD HISTORY DEFAULT PASSWORD REUSE INTERVAL DEFAULT PASSWORD REQUIRE CURRENT DEFAULT`; failed-login tracking and temporary account locking are disabled
- Account locking: `ACCOUNT UNLOCK`

An account when first created has no privileges and a default role of `NONE`. To assign privileges or roles, use the `GRANT` statement.

Each account name uses the format described in Section 6.2.4, “Specifying Account Names”. For example:

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

The host name part of the account name, if omitted, defaults to `'%'`.

Each *`user`* value naming an account may be followed by an optional *`auth_option`* value that indicates how the account authenticates. These values enable account authentication plugins and credentials (for example, a password) to be specified. Each *`auth_option`* value applies *only* to the account named immediately preceding it.

Following the *`user`* specifications, the statement may include options for SSL/TLS, resource-limit, password-management, and locking properties. All such options are *global* to the statement and apply to *all* accounts named in the statement.

Example: Create an account that uses the default authentication plugin and the given password. Mark the password expired so that the user must choose a new one at the first connection to the server:

```
CREATE USER 'jeffrey'@'localhost'
```

```
  IDENTIFIED BY 'new_password' PASSWORD EXPIRE;
```

Example: Create an account that uses the `caching_sha2_password` authentication plugin and the given password. Require that a new password be chosen every 180 days, and enable failed-login tracking, such that three consecutive incorrect passwords cause temporary account locking for two days:

```
CREATE USER 'jeffrey'@'localhost'
  IDENTIFIED WITH caching_sha2_password BY 'new_password'
  PASSWORD EXPIRE INTERVAL 180 DAY
  FAILED_LOGIN_ATTEMPTS 3 PASSWORD_LOCK_TIME 2;
```

Example: Create multiple accounts, specifying some per-account properties and some global properties:

```
CREATE USER
  'jeffrey'@'localhost' IDENTIFIED WITH mysql_native_password
                                    BY 'new_password1',
  'jeanne'@'localhost' IDENTIFIED WITH caching_sha2_password
                                   BY 'new_password2'
  REQUIRE X509 WITH MAX_QUERIES_PER_HOUR 60
  PASSWORD HISTORY 5
  ACCOUNT LOCK;
```

Each *`auth_option`* value (`IDENTIFIED WITH ... BY` in this case) applies only to the account named immediately preceding it, so each account uses the immediately following authentication plugin and password.

The remaining properties apply globally to all accounts named in the statement, so for both accounts:

- Connections must be made using a valid X.509 certificate.
- Up to 60 queries per hour are permitted.
- Password changes cannot reuse any of the five most recent passwords.
- The account is locked initially, so effectively it is a placeholder and cannot be used until an administrator unlocks it.

CREATE USER Authentication Options

An account name may be followed by an *`auth_option`* authentication option that specifies the account authentication plugin, credentials, or both.

Note

Prior to MySQL 8.0.27, *`auth_option`* defines the sole method by which an account authenticates. That is, all accounts use one-factor/single-factor authentication (1FA/SFA). MySQL 8.0.27 and higher supports multifactor authentication (MFA), such that accounts can have up to three authentication methods. That is, accounts can use two-factor authentication (2FA) or three-factor authentication (3FA). The syntax and semantics of *`auth_option`* remain unchanged, but *`auth_option`* may be followed by specifications for additional authentication methods. This section describes *`auth_option`*. For details about the optional MFA-related following clauses, see CREATE USER Multifactor Authentication Options.

Note

Clauses for random password generation apply only to accounts that use an authentication plugin that stores credentials internally to MySQL. For accounts that use a plugin that performs authentication against a credentials system that is external to MySQL, password management must be handled externally against that system as well. For more information about internal credentials storage, see Section 6.2.15, “Password Management”.

- *auth_plugin* names an authentication plugin. The plugin name can be a quoted string literal or an unquoted name. Plugin names are stored in the `plugin` column of the `mysql.user` system table.

  For *auth_option* syntax that does not specify an authentication plugin, the server assigns the default plugin, determined as described in The Default Authentication Plugin. For descriptions of each plugin, see Section 6.4.1, “Authentication Plugins”.

- Credentials that are stored internally are stored in the `mysql.user` system table. An `'auth_string'` value or `RANDOM PASSWORD` specifies account credentials, either as a cleartext (unencrypted) string or hashed in the format expected by the authentication plugin associated with the account, respectively:

  - For syntax that uses `BY 'auth_string'`, the string is cleartext and is passed to the authentication plugin for possible hashing. The result returned by the plugin is stored in the `mysql.user` table. A plugin may use the value as specified, in which case no hashing occurs.

  - For syntax that uses `BY RANDOM PASSWORD`, MySQL generates a random password and as cleartext and passes it to the authentication plugin for possible hashing. The result returned by the plugin is stored in the `mysql.user` table. A plugin may use the value as specified, in which case no hashing occurs.

    Randomly generated passwords are available as of MySQL 8.0.18 and have the characteristics described in Random Password Generation.

  - For syntax that uses `AS 'auth_string'`, the string is assumed to be already in the format the authentication plugin requires, and is stored as is in the `mysql.user` table. If a plugin requires a hashed value, the value must be already hashed in a format appropriate for the plugin; otherwise, the value cannot be used by the plugin and correct authentication of client connections does not occur.

    As of MySQL 8.0.17, a hashed string can be either a string literal or a hexadecimal value. The latter corresponds to the type of value displayed by `SHOW CREATE USER` for password hashes containing unprintable characters when the `print_identified_with_as_hex` system variable is enabled.

  - If an authentication plugin performs no hashing of the authentication string, the `BY 'auth_string'` and `AS 'auth_string'` clauses have the same effect: The authentication string is stored as is in the `mysql.user` system table.

`CREATE USER` permits these *auth_option* syntaxes:

- `IDENTIFIED BY 'auth_string'`

  Sets the account authentication plugin to the default plugin, passes the cleartext `'auth_string'` value to the plugin for possible hashing, and stores the result in the account row in the `mysql.user` system table.

- `IDENTIFIED BY RANDOM PASSWORD`

  Sets the account authentication plugin to the default plugin, generates a random password, passes the cleartext password value to the plugin for possible hashing, and stores the result in the account row in the `mysql.user` system table. The statement also returns the cleartext password in a result set to make it available to the user or application executing the statement. For details about the result set and characteristics of randomly generated passwords, see Random Password Generation.

- `IDENTIFIED WITH auth_plugin`

  Sets the account authentication plugin to *auth_plugin*, clears the credentials to the empty string, and stores the result in the account row in the `mysql.user` system table.

- `IDENTIFIED WITH auth_plugin BY 'auth_string'`

Sets the account authentication plugin to *auth_plugin*, passes the cleartext *'auth_string'* value to the plugin for possible hashing, and stores the result in the account row in the mysql.user system table.

- IDENTIFIED WITH *auth_plugin* BY RANDOM PASSWORD

  Sets the account authentication plugin to *auth_plugin*, generates a random password, passes the cleartext password value to the plugin for possible hashing, and stores the result in the account row in the mysql.user system table. The statement also returns the cleartext password in a result set to make it available to the user or application executing the statement. For details about the result set and characteristics of randomly generated passwords, see Random Password Generation.

- IDENTIFIED WITH *auth_plugin* AS *'auth_string'*

  Sets the account authentication plugin to *auth_plugin* and stores the *'auth_string'* value as is in the mysql.user account row. If the plugin requires a hashed string, the string is assumed to be already hashed in the format the plugin requires.

Example: Specify the password as cleartext; the default plugin is used:

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

Example: Specify the authentication plugin, along with a cleartext password value:

```
CREATE USER 'jeffrey'@'localhost'
  IDENTIFIED WITH mysql_native_password BY 'password';
```

In each case, the password value stored in the account row is the cleartext value *'password'* after it has been hashed by the authentication plugin associated with the account.

For additional information about setting passwords and authentication plugins, see Section 6.2.14, “Assigning Account Passwords”, and Section 6.2.17, “Pluggable Authentication”.

CREATE USER Multifactor Authentication Options

The *auth_option* part of CREATE USER defines an authentication method for one-factor/single-factor authentication (1FA/SFA). As of MySQL 8.0.27, CREATE USER has clauses that support multifactor authentication (MFA), such that accounts can have up to three authentication methods. That is, accounts can use two-factor authentication (2FA) or three-factor authentication (3FA).

The authentication_policy system variable defines constraints for CREATE USER statements with multifactor authentication (MFA) clauses. For example, the authentication_policy setting controls the number of authentication factors that accounts may have, and for each factor, which authentication methods are permitted. See Configuring the Multifactor Authentication Policy.

For information about factor-specific rules that determine the default authentication plugin for authentication clauses that name no plugin, see The Default Authentication Plugin.

Following *auth_option*, there may appear different optional MFA clauses:

- *2fa_auth_option*: Specifies a factor 2 authentication method. The following example defines caching_sha2_password as the factor 1 authentication method, and authentication_ldap_sasl as the factor 2 authentication method.

  ```
  CREATE USER 'u1'@'localhost'
    IDENTIFIED WITH caching_sha2_password
      BY 'sha2_password'
    AND IDENTIFIED WITH authentication_ldap_sasl
      AS 'uid=u1_ldap,ou=People,dc=example,dc=com';
  ```

- *3fa_auth_option*: Following *2fa_auth_option*, there may appear a *3fa_auth_option* clause to specify a factor 3 authentication method. The following example defines caching_sha2_password as the factor 1 authentication method, authentication_ldap_sasl

as the factor 2 authentication method, and `authentication_fido` as the factor 3 authentication method

```
CREATE USER 'u1'@'localhost'
  IDENTIFIED WITH caching_sha2_password
    BY 'sha2_password'
  AND IDENTIFIED WITH authentication_ldap_sasl
    AS 'uid=u1_ldap,ou=People,dc=example,dc=com'
  AND IDENTIFIED WITH authentication_fido;
```

- *`initial_auth_option`*: Specifies an initial authentication method for configuring FIDO passwordless authentication. As shown in the following, temporary authentication using either a generated random password or a user-specified *`auth-string`* is required to enable FIDO passwordless authentication.

  ```
  CREATE USER user
    IDENTIFIED WITH authentication_fido
    INITIAL AUTHENTICATION IDENTIFIED BY {RANDOM PASSWORD | 'auth_string'};
  ```

  For information about configuring passwordless authentication using FIDO pluggable authentication, See FIDO Passwordless Authentication.

CREATE USER Role Options

The `DEFAULT ROLE` clause defines which roles become active when the user connects to the server and authenticates, or when the user executes the `SET ROLE DEFAULT` statement during a session.

Each role name uses the format described in Section 6.2.5, “Specifying Role Names”. For example:

```
CREATE USER 'joe'@'10.0.0.1' DEFAULT ROLE administrator, developer;
```

The host name part of the role name, if omitted, defaults to `'%'`.

The `DEFAULT ROLE` clause permits a list of one or more comma-separated role names. These roles must exist at the time `CREATE USER` is executed; otherwise the statement raises an error (`ER_USER_DOES_NOT_EXIST`), and the user is not created.

CREATE USER SSL/TLS Options

MySQL can check X.509 certificate attributes in addition to the usual authentication that is based on the user name and credentials. For background information on the use of SSL/TLS with MySQL, see Section 6.3, “Using Encrypted Connections”.

To specify SSL/TLS-related options for a MySQL account, use a `REQUIRE` clause that specifies one or more *`tls_option`* values.

Order of `REQUIRE` options does not matter, but no option can be specified twice. The `AND` keyword is optional between `REQUIRE` options.

`CREATE USER` permits these *`tls_option`* values:

- `NONE`

  Indicates that all accounts named by the statement have no SSL or X.509 requirements. Unencrypted connections are permitted if the user name and password are valid. Encrypted connections can be used, at the client's option, if the client has the proper certificate and key files.

  ```
  CREATE USER 'jeffrey'@'localhost' REQUIRE NONE;
  ```

  Clients attempt to establish a secure connection by default. For clients that have `REQUIRE NONE`, the connection attempt falls back to an unencrypted connection if a secure connection cannot be established. To require an encrypted connection, a client need specify only the `--ssl-mode=REQUIRED` option; the connection attempt fails if a secure connection cannot be established.

  `NONE` is the default if no SSL-related `REQUIRE` options are specified.

- `SSL`

  Tells the server to permit only encrypted connections for all accounts named by the statement.

  ```
  CREATE USER 'jeffrey'@'localhost' REQUIRE SSL;
  ```

  Clients attempt to establish a secure connection by default. For accounts that have `REQUIRE SSL`, the connection attempt fails if a secure connection cannot be established.

- `X509`

  For all accounts named by the statement, requires that clients present a valid certificate, but the exact certificate, issuer, and subject do not matter. The only requirement is that it should be possible to verify its signature with one of the CA certificates. Use of X.509 certificates always implies encryption, so the `SSL` option is unnecessary in this case.

  ```
  CREATE USER 'jeffrey'@'localhost' REQUIRE X509;
  ```

  For accounts with `REQUIRE X509`, clients must specify the `--ssl-key` and `--ssl-cert` options to connect. (It is recommended but not required that `--ssl-ca` also be specified so that the public certificate provided by the server can be verified.) This is true for `ISSUER` and `SUBJECT` as well because those `REQUIRE` options imply the requirements of `X509`.

- `ISSUER 'issuer'`

  For all accounts named by the statement, requires that clients present a valid X.509 certificate issued by CA `'issuer'`. If a client presents a certificate that is valid but has a different issuer, the server rejects the connection. Use of X.509 certificates always implies encryption, so the `SSL` option is unnecessary in this case.

  ```
  CREATE USER 'jeffrey'@'localhost'
    REQUIRE ISSUER '/C=SE/ST=Stockholm/L=Stockholm/
      O=MySQL/CN=CA/emailAddress=ca@example.com';
  ```

  Because `ISSUER` implies the requirements of `X509`, clients must specify the `--ssl-key` and `--ssl-cert` options to connect. (It is recommended but not required that `--ssl-ca` also be specified so that the public certificate provided by the server can be verified.)

- `SUBJECT 'subject'`

  For all accounts named by the statement, requires that clients present a valid X.509 certificate containing the subject *`subject`*. If a client presents a certificate that is valid but has a different subject, the server rejects the connection. Use of X.509 certificates always implies encryption, so the `SSL` option is unnecessary in this case.

  ```
  CREATE USER 'jeffrey'@'localhost'
    REQUIRE SUBJECT '/C=SE/ST=Stockholm/L=Stockholm/
      O=MySQL demo client certificate/
      CN=client/emailAddress=client@example.com';
  ```

  MySQL does a simple string comparison of the `'subject'` value to the value in the certificate, so lettercase and component ordering must be given exactly as present in the certificate.

  Because `SUBJECT` implies the requirements of `X509`, clients must specify the `--ssl-key` and `--ssl-cert` options to connect. (It is recommended but not required that `--ssl-ca` also be specified so that the public certificate provided by the server can be verified.)

- `CIPHER 'cipher'`

  For all accounts named by the statement, requires a specific cipher method for encrypting connections. This option is needed to ensure that ciphers and key lengths of sufficient strength are used. Encryption can be weak if old algorithms using short encryption keys are used.

  ```
  CREATE USER 'jeffrey'@'localhost'
  ```

```
  REQUIRE CIPHER 'EDH-RSA-DES-CBC3-SHA';
```

The `SUBJECT`, `ISSUER`, and `CIPHER` options can be combined in the `REQUIRE` clause:

```
CREATE USER 'jeffrey'@'localhost'
  REQUIRE SUBJECT '/C=SE/ST=Stockholm/L=Stockholm/
    O=MySQL demo client certificate/
    CN=client/emailAddress=client@example.com'
  AND ISSUER '/C=SE/ST=Stockholm/L=Stockholm/
    O=MySQL/CN=CA/emailAddress=ca@example.com'
  AND CIPHER 'EDH-RSA-DES-CBC3-SHA';
```

CREATE USER Resource-Limit Options

It is possible to place limits on use of server resources by an account, as discussed in Section 6.2.21, “Setting Account Resource Limits”. To do so, use a `WITH` clause that specifies one or more *`resource_option`* values.

Order of `WITH` options does not matter, except that if a given resource limit is specified multiple times, the last instance takes precedence.

`CREATE USER` permits these *`resource_option`* values:

- `MAX_QUERIES_PER_HOUR` *`count`*, `MAX_UPDATES_PER_HOUR` *`count`*, `MAX_CONNECTIONS_PER_HOUR` *`count`*

  For all accounts named by the statement, these options restrict how many queries, updates, and connections to the server are permitted to each account during any given one-hour period. If *`count`* is `0` (the default), this means that there is no limitation for the account.

- `MAX_USER_CONNECTIONS` *`count`*

  For all accounts named by the statement, restricts the maximum number of simultaneous connections to the server by each account. A nonzero *`count`* specifies the limit for the account explicitly. If *`count`* is `0` (the default), the server determines the number of simultaneous connections for the account from the global value of the `max_user_connections` system variable. If `max_user_connections` is also zero, there is no limit for the account.

Example:

```
CREATE USER 'jeffrey'@'localhost'
  WITH MAX_QUERIES_PER_HOUR 500 MAX_UPDATES_PER_HOUR 100;
```

CREATE USER Password-Management Options

`CREATE USER` supports several *`password_option`* values for password management:

- Password expiration options: You can expire an account password manually and establish its password expiration policy. Policy options do not expire the password. Instead, they determine how the server applies automatic expiration to the account based on password age, which is assessed from the date and time of the most recent account password change.

- Password reuse options: You can restrict password reuse based on number of password changes, time elapsed, or both.

- Password verification-required options: You can indicate whether attempts to change an account password must specify the current password, as verification that the user attempting to make the change actually knows the current password.

- Incorrect-password failed-login tracking options: You can cause the server to track failed login attempts and temporarily lock accounts for which too many consecutive incorrect passwords are given. The required number of failures and the lock time are configurable.

This section describes the syntax for password-management options. For information about establishing policy for password management, see Section 6.2.15, “Password Management”.

If multiple password-management options of a given type are specified, the last one takes precedence. For example, `PASSWORD EXPIRE DEFAULT PASSWORD EXPIRE NEVER` is the same as `PASSWORD EXPIRE NEVER`.

Note

Except for the options that pertain to failed-login tracking, password-management options apply only to accounts that use an authentication plugin that stores credentials internally to MySQL. For accounts that use a plugin that performs authentication against a credentials system that is external to MySQL, password management must be handled externally against that system as well. For more information about internal credentials storage, see Section 6.2.15, “Password Management”.

A client has an expired password if the account password was expired manually or the password age is considered greater than its permitted lifetime per the automatic expiration policy. In this case, the server either disconnects the client or restricts the operations permitted to it (see Section 6.2.16, “Server Handling of Expired Passwords”). Operations performed by a restricted client result in an error until the user establishes a new account password.

`CREATE USER` permits these *`password_option`* values for controlling password expiration:

- `PASSWORD EXPIRE`

  Immediately marks the password expired for all accounts named by the statement.

  ```
  CREATE USER 'jeffrey'@'localhost' PASSWORD EXPIRE;
  ```

- `PASSWORD EXPIRE DEFAULT`

  Sets all accounts named by the statement so that the global expiration policy applies, as specified by the `default_password_lifetime` system variable.

  ```
  CREATE USER 'jeffrey'@'localhost' PASSWORD EXPIRE DEFAULT;
  ```

- `PASSWORD EXPIRE NEVER`

  This expiration option overrides the global policy for all accounts named by the statement. For each, it disables password expiration so that the password never expires.

  ```
  CREATE USER 'jeffrey'@'localhost' PASSWORD EXPIRE NEVER;
  ```

- `PASSWORD EXPIRE INTERVAL N DAY`

  This expiration option overrides the global policy for all accounts named by the statement. For each, it sets the password lifetime to *`N`* days. The following statement requires the password to be changed every 180 days:

  ```
  CREATE USER 'jeffrey'@'localhost' PASSWORD EXPIRE INTERVAL 180 DAY;
  ```

`CREATE USER` permits these *`password_option`* values for controlling reuse of previous passwords based on required minimum number of password changes:

- `PASSWORD HISTORY DEFAULT`

  Sets all accounts named by the statement so that the global policy about password history length applies, to prohibit reuse of passwords before the number of changes specified by the `password_history` system variable.

  ```
  CREATE USER 'jeffrey'@'localhost' PASSWORD HISTORY DEFAULT;
  ```

- `PASSWORD HISTORY N`

  This history-length option overrides the global policy for all accounts named by the statement. For each, it sets the password history length to *`N`* passwords, to prohibit reusing any of the *`N`* most recently chosen passwords. The following statement prohibits reuse of any of the previous 6 passwords:

```
CREATE USER 'jeffrey'@'localhost' PASSWORD HISTORY 6;
```

`CREATE USER` permits these *`password_option`* values for controlling reuse of previous passwords based on time elapsed:

- `PASSWORD REUSE INTERVAL DEFAULT`

  Sets all statements named by the account so that the global policy about time elapsed applies, to prohibit reuse of passwords newer than the number of days specified by the `password_reuse_interval` system variable.

```
CREATE USER 'jeffrey'@'localhost' PASSWORD REUSE INTERVAL DEFAULT;
```

- `PASSWORD REUSE INTERVAL N DAY`

  This time-elapsed option overrides the global policy for all accounts named by the statement. For each, it sets the password reuse interval to *`N`* days, to prohibit reuse of passwords newer than that many days. The following statement prohibits password reuse for 360 days:

```
CREATE USER 'jeffrey'@'localhost' PASSWORD REUSE INTERVAL 360 DAY;
```

`CREATE USER` permits these *`password_option`* values for controlling whether attempts to change an account password must specify the current password, as verification that the user attempting to make the change actually knows the current password:

- `PASSWORD REQUIRE CURRENT`

  This verification option overrides the global policy for all accounts named by the statement. For each, it requires that password changes specify the current password.

```
CREATE USER 'jeffrey'@'localhost' PASSWORD REQUIRE CURRENT;
```

- `PASSWORD REQUIRE CURRENT OPTIONAL`

  This verification option overrides the global policy for all accounts named by the statement. For each, it does not require that password changes specify the current password. (The current password may but need not be given.)

```
CREATE USER 'jeffrey'@'localhost' PASSWORD REQUIRE CURRENT OPTIONAL;
```

- `PASSWORD REQUIRE CURRENT DEFAULT`

  Sets all statements named by the account so that the global policy about password verification applies, as specified by the `password_require_current` system variable.

```
CREATE USER 'jeffrey'@'localhost' PASSWORD REQUIRE CURRENT DEFAULT;
```

As of MySQL 8.0.19, `CREATE USER` permits these *`password_option`* values for controlling failed-login tracking:

- `FAILED_LOGIN_ATTEMPTS N`

  Whether to track account login attempts that specify an incorrect password. *`N`* must be a number from 0 to 32767. A value of 0 disables failed-login tracking. Values greater than 0 indicate how many consecutive password failures cause temporary account locking (if `PASSWORD_LOCK_TIME` is also nonzero).

- `PASSWORD_LOCK_TIME {N | UNBOUNDED}`

  How long to lock the account after too many consecutive login attempts provide an incorrect password. *`N`* must be a number from 0 to 32767, or `UNBOUNDED`. A value of 0 disables temporary account locking. Values greater than 0 indicate how long to lock the account in days. A value of `UNBOUNDED` causes the account locking duration to be unbounded; once locked, the account remains in a locked state until unlocked. For information about the conditions under which unlocking occurs, see Failed-Login Tracking and Temporary Account Locking.

For failed-login tracking and temporary locking to occur, an account's `FAILED_LOGIN_ATTEMPTS` and `PASSWORD_LOCK_TIME` options both must be nonzero. The following statement creates an account that remains locked for two days after four consecutive password failures:

```
CREATE USER 'jeffrey'@'localhost'
  FAILED_LOGIN_ATTEMPTS 4 PASSWORD_LOCK_TIME 2;
```

CREATE USER Comment and Attribute Options

As of MySQL 8.0.21, you can create an account with an optional comment or attribute, as described here:

- **User comment**

  To set a user comment, add `COMMENT 'user_comment'` to the `CREATE USER` statement, where *`user_comment`* is the text of the user comment.

  Example (omitting any other options):

  ```
  CREATE USER 'jon'@'localhost' COMMENT 'Some information about Jon';
  ```

- **User attribute**

  A user attribute is a JSON object made up of one or more key-value pairs, and is set by including `ATTRIBUTE 'json_object'` as part of `CREATE USER`. *`json_object`* must be a valid JSON object.

  Example (omitting any other options):

  ```
  CREATE USER 'jim'@'localhost'
      ATTRIBUTE '{"fname": "James", "lname": "Scott", "phone": "123-456-7890"}';
  ```

User comments and user attributes are stored together in the `ATTRIBUTE` column of the Information Schema `USER_ATTRIBUTES` table. This query displays the row in this table inserted by the statement just shown for creating the user `jim@localhost`:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.USER_ATTRIBUTES
    ->     WHERE USER = 'jim' AND HOST = 'localhost'\G
*************************** 1. row ***************************
     USER: jim
     HOST: localhost
ATTRIBUTE: {"fname": "James", "lname": "Scott", "phone": "123-456-7890"}
1 row in set (0.00 sec)
```

The `COMMENT` option in actuality provides a shortcut for setting a user attribute whose only element has `comment` as its key and whose value is the argument supplied for the option. You can see this by executing the statement `CREATE USER 'jon'@'localhost' COMMENT 'Some information about Jon'`, and observing the row which it inserts into the `USER_ATTRIBUTES` table:

```
mysql> CREATE USER 'jon'@'localhost' COMMENT 'Some information about Jon';
Query OK, 0 rows affected (0.06 sec)

mysql> SELECT * FROM INFORMATION_SCHEMA.USER_ATTRIBUTES
    ->     WHERE USER = 'jon' AND HOST = 'localhost';
+------+-----------+----------------------------------------+
| USER | HOST      | ATTRIBUTE                              |
```

```
+------+-----------+---------------------------------------+
| jon  | localhost | {"comment": "Some information about Jon"} |
+------+-----------+---------------------------------------+
1 row in set (0.00 sec)
```

You cannot use `COMMENT` and `ATTRIBUTE` together in the same `CREATE USER` statement; attempting to do so causes a syntax error. To set a user comment concurrently with setting a user attribute, use `ATTRIBUTE` and include in its argument a value with a `comment` key, like this:

```
mysql> CREATE USER 'bill'@'localhost'
    ->        ATTRIBUTE '{"fname":"William", "lname":"Schmidt",
    ->        "comment":"Website developer"}';
Query OK, 0 rows affected (0.16 sec)
```

Since the content of the `ATTRIBUTE` row is a JSON object, you can employ any appropriate MySQL JSON functions or operators to manipulate it, as shown here:

```
mysql> SELECT
    ->   USER AS User,
    ->   HOST AS Host,
    ->   CONCAT(ATTRIBUTE->>"$.fname"," ",ATTRIBUTE->>"$.lname") AS 'Full Name',
    ->   ATTRIBUTE->>"$.comment" AS Comment
    -> FROM INFORMATION_SCHEMA.USER_ATTRIBUTES
    -> WHERE USER='bill' AND HOST='localhost';
+------+-----------+-----------------+-------------------+
| User | Host      | Full Name       | Comment           |
+------+-----------+-----------------+-------------------+
| bill | localhost | William Schmidt | Website developer |
+------+-----------+-----------------+-------------------+
1 row in set (0.00 sec)
```

To set or to make changes in the user comment or user attribute for an existing user, you can use a `COMMENT` or `ATTRIBUTE` option with an `ALTER USER` statement.

Because the user comment and user attribute are stored together internally in a single `JSON` column, this sets an upper limit on their maximum combined size; see JSON Storage Requirements, for more information.

See also the description of the Information Schema `USER_ATTRIBUTES` table for more information and examples.

CREATE USER Account-Locking Options

MySQL supports account locking and unlocking using the `ACCOUNT LOCK` and `ACCOUNT UNLOCK` options, which specify the locking state for an account. For additional discussion, see Section 6.2.20, “Account Locking”.

If multiple account-locking options are specified, the last one takes precedence.

CREATE USER Binary Logging

`CREATE USER` is written to the binary log if it succeeds, but not if it fails; in that case, rollback occurs and no changes are made. A statement written to the binary log includes all named users. If the `IF NOT EXISTS` clause is given, this includes even users that already exist and were not created.

The statement written to the binary log specifies an authentication plugin for each user, determined as follows:

- The plugin named in the original statement, if one was specified.
- Otherwise, the default authentication plugin. In particular, if a user `u1` already exists and uses a nondefault authentication plugin, the statement written to the binary log for `CREATE USER IF NOT EXISTS u1` names the default authentication plugin. (If the statement written to the binary log must specify a nondefault authentication plugin for a user, include it in the original statement.)

If the server adds the default authentication plugin for any nonexisting users in the statement written to the binary log, it writes a warning to the error log naming those users.

If the original statement specifies the `FAILED_LOGIN_ATTEMPTS` or `PASSWORD_LOCK_TIME` option, the statement written to the binary log includes the option.

`CREATE USER` statements with clauses that support multifactor authentication (MFA) are written to the binary log.

- `CREATE USER ... IDENTIFIED WITH .. INITIAL AUTHENTICATION IDENTIFIED WITH ...` statements are written to the binary log as `CREATE USER .. IDENTIFIED WITH .. INITIAL AUTHENTICATION IDENTIFIED WITH .. AS '`*`password-hash`*`'`, where the *`password-hash`* is the user-specified *`auth-string`* or the random password generated by server when the `RANDOM PASSWORD` clause is specified.

13.7.1.4 DROP ROLE Statement

```
DROP ROLE [IF EXISTS] role [, role ] ...
```

`DROP ROLE` removes one or more roles (named collections of privileges). To use this statement, you must have the global `DROP ROLE` or `CREATE USER` privilege. When the `read_only` system variable is enabled, `DROP ROLE` additionally requires the `CONNECTION_ADMIN` privilege (or the deprecated `SUPER` privilege).

As of MySQL 8.0.16, users who have the `CREATE USER` privilege can use this statement to drop accounts that are locked or unlocked. Users who have the `DROP ROLE` privilege can use this statement only to drop accounts that are locked (unlocked accounts are presumably user accounts used to log in to the server and not just as roles).

Roles named in the `mandatory_roles` system variable value cannot be dropped.

`DROP ROLE` either succeeds for all named roles or rolls back and has no effect if any error occurs. By default, an error occurs if you try to drop a role that does not exist. If the `IF EXISTS` clause is given, the statement produces a warning for each named role that does not exist, rather than an error.

The statement is written to the binary log if it succeeds, but not if it fails; in that case, rollback occurs and no changes are made. A statement written to the binary log includes all named roles. If the `IF EXISTS` clause is given, this includes even roles that do not exist and were not dropped.

Each role name uses the format described in Section 6.2.5, “Specifying Role Names”. For example:

```
DROP ROLE 'admin', 'developer';
DROP ROLE 'webapp'@'localhost';
```

The host name part of the role name, if omitted, defaults to `'%'`.

A dropped role is automatically revoked from any user account (or role) to which the role was granted. Within any current session for such an account, its adjusted privileges apply beginning with the next statement executed.

For role usage examples, see Section 6.2.10, “Using Roles”.

13.7.1.5 DROP USER Statement

```
DROP USER [IF EXISTS] user [, user] ...
```

The `DROP USER` statement removes one or more MySQL accounts and their privileges. It removes privilege rows for the account from all grant tables.

Roles named in the `mandatory_roles` system variable value cannot be dropped.

To use `DROP USER`, you must have the global `CREATE USER` privilege, or the `DELETE` privilege for the `mysql` system schema. When the `read_only` system variable is enabled, `DROP USER` additionally requires the `CONNECTION_ADMIN` privilege (or the deprecated `SUPER` privilege).

As of MySQL 8.0.22, `DROP USER` fails with an error if any account to be dropped is named as the `DEFINER` attribute for any stored object. (That is, the statement fails if dropping an account would cause a stored object to become orphaned.) To perform the operation anyway, you must have the `SET_USER_ID` privilege; in this case, the statement succeeds with a warning rather than failing with an error. For additional information, including how to identify which objects name a given account as the `DEFINER` attribute, see Orphan Stored Objects.

`DROP USER` either succeeds for all named users or rolls back and has no effect if any error occurs. By default, an error occurs if you try to drop a user that does not exist. If the `IF EXISTS` clause is given, the statement produces a warning for each named user that does not exist, rather than an error.

The statement is written to the binary log if it succeeds, but not if it fails; in that case, rollback occurs and no changes are made. A statement written to the binary log includes all named users. If the `IF EXISTS` clause is given, this includes even users that do not exist and were not dropped.

Each account name uses the format described in Section 6.2.4, “Specifying Account Names”. For example:

```
DROP USER 'jeffrey'@'localhost';
```

The host name part of the account name, if omitted, defaults to `'%'`.

Important

`DROP USER` does not automatically close any open user sessions. Rather, in the event that a user with an open session is dropped, the statement does not take effect until that user's session is closed. Once the session is closed, the user is dropped, and that user's next attempt to log in fails. *This is by design*.

`DROP USER` does not automatically drop or invalidate databases or objects within them that the old user created. This includes stored programs or views for which the `DEFINER` attribute names the dropped user. Attempts to access such objects may produce an error if they execute in definer security context. (For information about security context, see Section 25.6, “Stored Object Access Control”.)

13.7.1.6 GRANT Statement

```
GRANT
    priv_type [(column_list)]
      [, priv_type [(column_list)]] ...
    ON [object_type] priv_level
    TO user_or_role [, user_or_role] ...
    [WITH GRANT OPTION]
    [AS user
        [WITH ROLE
            DEFAULT
          | NONE
          | ALL
          | ALL EXCEPT role [, role ] ...
          | role [, role ] ...
        ]
    ]
}

GRANT PROXY ON user_or_role
    TO user_or_role [, user_or_role] ...
    [WITH GRANT OPTION]

GRANT role [, role] ...
    TO user_or_role [, user_or_role] ...
    [WITH ADMIN OPTION]

object_type: {
    TABLE
  | FUNCTION
  | PROCEDURE
```

```
}

priv_level: {
    *
  | *.*
  | db_name.*
  | db_name.tbl_name
  | tbl_name
  | db_name.routine_name
}

user_or_role: {
    user (see Section 6.2.4, “Specifying Account Names”)
  | role (see Section 6.2.5, “Specifying Role Names”)
}
```

The `GRANT` statement assigns privileges and roles to MySQL user accounts and roles. There are several aspects to the `GRANT` statement, described under the following topics:

- GRANT General Overview
- Object Quoting Guidelines
- Account Names
- Privileges Supported by MySQL
- Global Privileges
- Database Privileges
- Table Privileges
- Column Privileges
- Stored Routine Privileges
- Proxy User Privileges
- Granting Roles
- The `AS` Clause and Privilege Restrictions
- Other Account Characteristics
- MySQL and Standard SQL Versions of GRANT

GRANT General Overview

The `GRANT` statement enables system administrators to grant privileges and roles, which can be granted to user accounts and roles. These syntax restrictions apply:

- `GRANT` cannot mix granting both privileges and roles in the same statement. A given `GRANT` statement must grant either privileges or roles.
- The `ON` clause distinguishes whether the statement grants privileges or roles:
  - With `ON`, the statement grants privileges.
  - Without `ON`, the statement grants roles.
  - It is permitted to assign both privileges and roles to an account, but you must use separate `GRANT` statements, each with syntax appropriate to what is to be granted.

For more information about roles, see Section 6.2.10, “Using Roles”.

To grant a privilege with `GRANT`, you must have the `GRANT OPTION` privilege, and you must have the privileges that you are granting. (Alternatively, if you have the `UPDATE` privilege for the grant tables in the `mysql` system schema, you can grant any account any privilege.) When the `read_only` system variable is enabled, `GRANT` additionally requires the `CONNECTION_ADMIN` privilege (or the deprecated `SUPER` privilege).

`GRANT` either succeeds for all named users and roles or rolls back and has no effect if any error occurs. The statement is written to the binary log only if it succeeds for all named users and roles.

The `REVOKE` statement is related to `GRANT` and enables administrators to remove account privileges. See Section 13.7.1.8, “REVOKE Statement”.

Each account name uses the format described in Section 6.2.4, “Specifying Account Names”. Each role name uses the format described in Section 6.2.5, “Specifying Role Names”. For example:

```
GRANT ALL ON db1.* TO 'jeffrey'@'localhost';
GRANT 'role1', 'role2' TO 'user1'@'localhost', 'user2'@'localhost';
GRANT SELECT ON world.* TO 'role3';
```

The host name part of the account or role name, if omitted, defaults to `'%'`.

Normally, a database administrator first uses `CREATE USER` to create an account and define its nonprivilege characteristics such as its password, whether it uses secure connections, and limits on access to server resources, then uses `GRANT` to define its privileges. `ALTER USER` may be used to change the nonprivilege characteristics of existing accounts. For example:

```
CREATE USER 'jeffrey'@'localhost' IDENTIFIED BY 'password';
GRANT ALL ON db1.* TO 'jeffrey'@'localhost';
GRANT SELECT ON db2.invoice TO 'jeffrey'@'localhost';
ALTER USER 'jeffrey'@'localhost' WITH MAX_QUERIES_PER_HOUR 90;
```

From the `mysql` program, `GRANT` responds with `Query OK, 0 rows affected` when executed successfully. To determine what privileges result from the operation, use `SHOW GRANTS`. See Section 13.7.7.21, “SHOW GRANTS Statement”.

Important

Under some circumstances, `GRANT` may be recorded in server logs or on the client side in a history file such as `~/.mysql_history`, which means that cleartext passwords may be read by anyone having read access to that information. For information about the conditions under which this occurs for the server logs and how to control it, see Section 6.1.2.3, “Passwords and Logging”. For similar information about client-side logging, see Section 4.5.1.3, “mysql Client Logging”.

`GRANT` supports host names up to 255 characters long (60 characters prior to MySQL 8.0.17). User names can be up to 32 characters. Database, table, column, and routine names can be up to 64 characters.

Warning

Do not attempt to change the permissible length for user names by altering the `mysql.user` system table. Doing so results in unpredictable behavior which may even make it impossible for users to log in to the MySQL server. Never alter the structure of tables in the `mysql` system schema in any manner except by means of the procedure described in Section 2.10, “Upgrading MySQL”.

Object Quoting Guidelines

Several objects within `GRANT` statements are subject to quoting, although quoting is optional in many cases: Account, role, database, table, column, and routine names. For example, if a *`user_name`* or *`host_name`* value in an account name is legal as an unquoted identifier, you need not quote it.

However, quotation marks are necessary to specify a *`user_name`* string containing special characters (such as `-`), or a *`host_name`* string containing special characters or wildcard characters such as `%` (for example, `'test-user'@'%.com'`). Quote the user name and host name separately.

To specify quoted values:

- Quote database, table, column, and routine names as identifiers.
- Quote user names and host names as identifiers or as strings.
- Quote passwords as strings.

For string-quoting and identifier-quoting guidelines, see Section 9.1.1, “String Literals”, and Section 9.2, “Schema Object Names”.

The `_` and `%` wildcards are permitted when specifying database names in `GRANT` statements that grant privileges at the database level (`GRANT ... ON db_name.*`). This means, for example, that to use a `_` character as part of a database name, specify it using the `\` escape character as `\_` in the `GRANT` statement, to prevent the user from being able to access additional databases matching the wildcard pattern (for example, `GRANT ... ON `foo_bar`.* TO ...`).

Issuing multiple `GRANT` statements containing wildcards may not have the expected effect on DML statements; when resolving grants involving wildcards, MySQL takes only the first matching grant into consideration. In other words, if a user has two database-level grants using wildcards that match the same database, the grant which was created first is applied. Consider the database `db` and table `t` created using the statements shown here:

```
mysql> CREATE DATABASE db;
Query OK, 1 row affected (0.01 sec)

mysql> CREATE TABLE db.t (c INT);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO db.t VALUES ROW(1);
Query OK, 1 row affected (0.00 sec)
```

Next (assuming that the current account is the MySQL `root` account or another account having the necessary privileges), we create a user `u` then issue two `GRANT` statements containing wildcards, like this:

```
mysql> CREATE USER u;
Query OK, 0 rows affected (0.01 sec)

mysql> GRANT SELECT ON `d_`.* TO u;
Query OK, 0 rows affected (0.01 sec)

mysql> GRANT INSERT ON `d%`.* TO u;
Query OK, 0 rows affected (0.00 sec)

mysql> EXIT
```

```
Bye
```

If we end the session and then log in again with the `mysql` client, this time as `u`, we see that this account has only the privilege provided by the first matching grant, but not the second:

```
$> mysql -uu -hlocalhost
```

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 10
Server version: 8.0.34-tr Source distribution

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql> TABLE db.t;
+------+
| c    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

mysql> INSERT INTO db.t VALUES ROW(2);
ERROR 1142 (42000): INSERT command denied to user 'u'@'localhost' for table 't'
```

In privilege assignments, MySQL interprets occurrences of unescaped `_` and `%` SQL wildcard characters in database names as literal characters under these circumstances:

- When a database name is not used to grant privileges at the database level, but as a qualifier for granting privileges to some other object such as a table or routine (for example, `GRANT ... ON db_name.tbl_name`).
- Enabling `partial_revokes` causes MySQL to interpret unescaped `_` and `%` wildcard characters in database names as literal characters, just as if they had been escaped as `\_` and `\%`. Because this changes how MySQL interprets privileges, it may be advisable to avoid unescaped wildcard characters in privilege assignments for installations where `partial_revokes` may be enabled. For more information, see Section 6.2.12, “Privilege Restriction Using Partial Revokes”.

Account Names

A *`user`* value in a `GRANT` statement indicates a MySQL account to which the statement applies. To accommodate granting rights to users from arbitrary hosts, MySQL supports specifying the *`user`* value in the form `'user_name'@'host_name'`.

You can specify wildcards in the host name. For example, `'user_name'@'%.example.com'` applies to *`user_name`* for any host in the `example.com` domain, and `'user_name'@'198.51.100.%'` applies to *`user_name`* for any host in the `198.51.100` class C subnet.

The simple form `'user_name'` is a synonym for `'user_name'@'%'`.

MySQL does not support wildcards in user names. To refer to an anonymous user, specify an account with an empty user name with the `GRANT` statement:

```
GRANT ALL ON test.* TO ''@'localhost' ...;
```

In this case, any user who connects from the local host with the correct password for the anonymous user is permitted access, with the privileges associated with the anonymous-user account.

For additional information about user name and host name values in account names, see Section 6.2.4, “Specifying Account Names”.

Warning

If you permit local anonymous users to connect to the MySQL server, you should also grant privileges to all local users as `'user_name'@'localhost'`. Otherwise, the anonymous user account for `localhost` in the `mysql.user` system table is used when named users try to log in to the MySQL server from the local machine. For details, see Section 6.2.6, “Access Control, Stage 1: Connection Verification”.

To determine whether this issue applies to you, execute the following query, which lists any anonymous users:

```
SELECT Host, User FROM mysql.user WHERE User='';
```

> To avoid the problem just described, delete the local anonymous user account using this statement:
>
> ```
> DROP USER ''@'localhost';
> ```

Privileges Supported by MySQL

The following tables summarize the permissible static and dynamic *`priv_type`* privilege types that can be specified for the `GRANT` and `REVOKE` statements, and the levels at which each privilege can be granted. For additional information about each privilege, see Section 6.2.2, “Privileges Provided by MySQL”. For information about the differences between static and dynamic privileges, see Static Versus Dynamic Privileges.

Table 13.11 Permissible Static Privileges for GRANT and REVOKE

Privilege	Meaning and Grantable Levels
`ALL [PRIVILEGES]`	Grant all privileges at specified access level except `GRANT OPTION` and `PROXY`.
`ALTER`	Enable use of `ALTER TABLE`. Levels: Global, database, table.
`ALTER ROUTINE`	Enable stored routines to be altered or dropped. Levels: Global, database, routine.
`CREATE`	Enable database and table creation. Levels: Global, database, table.
`CREATE ROLE`	Enable role creation. Level: Global.
`CREATE ROUTINE`	Enable stored routine creation. Levels: Global, database.
`CREATE TABLESPACE`	Enable tablespaces and log file groups to be created, altered, or dropped. Level: Global.
`CREATE TEMPORARY TABLES`	Enable use of `CREATE TEMPORARY TABLE`. Levels: Global, database.
`CREATE USER`	Enable use of `CREATE USER`, `DROP USER`, `RENAME USER`, and `REVOKE ALL PRIVILEGES`. Level: Global.
`CREATE VIEW`	Enable views to be created or altered. Levels: Global, database, table.
`DELETE`	Enable use of `DELETE`. Level: Global, database, table.
`DROP`	Enable databases, tables, and views to be dropped. Levels: Global, database, table.
`DROP ROLE`	Enable roles to be dropped. Level: Global.
`EVENT`	Enable use of events for the Event Scheduler. Levels: Global, database.
`EXECUTE`	Enable the user to execute stored routines. Levels: Global, database, routine.
`FILE`	Enable the user to cause the server to read or write files. Level: Global.
`GRANT OPTION`	Enable privileges to be granted to or removed from other accounts. Levels: Global, database, table, routine, proxy.
`INDEX`	Enable indexes to be created or dropped. Levels: Global, database, table.

Privilege	Meaning and Grantable Levels
INSERT	Enable use of INSERT. Levels: Global, database, table, column.
LOCK TABLES	Enable use of LOCK TABLES on tables for which you have the SELECT privilege. Levels: Global, database.
PROCESS	Enable the user to see all processes with SHOW PROCESSLIST. Level: Global.
PROXY	Enable user proxying. Level: From user to user.
REFERENCES	Enable foreign key creation. Levels: Global, database, table, column.
RELOAD	Enable use of FLUSH operations. Level: Global.
REPLICATION CLIENT	Enable the user to ask where source or replica servers are. Level: Global.
REPLICATION SLAVE	Enable replicas to read binary log events from the source. Level: Global.
SELECT	Enable use of SELECT. Levels: Global, database, table, column.
SHOW DATABASES	Enable SHOW DATABASES to show all databases. Level: Global.
SHOW VIEW	Enable use of SHOW CREATE VIEW. Levels: Global, database, table.
SHUTDOWN	Enable use of mysqladmin shutdown. Level: Global.
SUPER	Enable use of other administrative operations such as CHANGE REPLICATION SOURCE TO, CHANGE MASTER TO, KILL, PURGE BINARY LOGS, SET GLOBAL, and mysqladmin debug command. Level: Global.
TRIGGER	Enable trigger operations. Levels: Global, database, table.
UPDATE	Enable use of UPDATE. Levels: Global, database, table, column.
USAGE	Synonym for “no privileges”

Table 13.12 Permissible Dynamic Privileges for GRANT and REVOKE

Privilege	Meaning and Grantable Levels
APPLICATION_PASSWORD_ADMIN	Enable dual password administration. Level: Global.
AUDIT_ABORT_EXEMPT	Allow queries blocked by audit log filter. Level: Global.
AUDIT_ADMIN	Enable audit log configuration. Level: Global.
AUTHENTICATION_POLICY_ADMIN	Enable authentication policy administration. Level: Global.
BACKUP_ADMIN	Enable backup administration. Level: Global.
BINLOG_ADMIN	Enable binary log control. Level: Global.
BINLOG_ENCRYPTION_ADMIN	Enable activation and deactivation of binary log encryption. Level: Global.

Privilege	Meaning and Grantable Levels
CLONE_ADMIN	Enable clone administration. Level: Global.
CONNECTION_ADMIN	Enable connection limit/restriction control. Level: Global.
ENCRYPTION_KEY_ADMIN	Enable InnoDB key rotation. Level: Global.
FIREWALL_ADMIN	Enable firewall rule administration, any user. Level: Global.
FIREWALL_EXEMPT	Exempt user from firewall restrictions. Level: Global.
FIREWALL_USER	Enable firewall rule administration, self. Level: Global.
FLUSH_OPTIMIZER_COSTS	Enable optimizer cost reloading. Level: Global.
FLUSH_STATUS	Enable status indicator flushing. Level: Global.
FLUSH_TABLES	Enable table flushing. Level: Global.
FLUSH_USER_RESOURCES	Enable user-resource flushing. Level: Global.
GROUP_REPLICATION_ADMIN	Enable Group Replication control. Level: Global.
INNODB_REDO_LOG_ARCHIVE	Enable redo log archiving administration. Level: Global.
INNODB_REDO_LOG_ENABLE	Enable or disable redo logging. Level: Global.
NDB_STORED_USER	Enable sharing of user or role between SQL nodes (NDB Cluster). Level: Global.
PASSWORDLESS_USER_ADMIN	Enable passwordless user account administration. Level: Global.
PERSIST_RO_VARIABLES_ADMIN	Enable persisting read-only system variables. Level: Global.
REPLICATION_APPLIER	Act as the PRIVILEGE_CHECKS_USER for a replication channel. Level: Global.
REPLICATION_SLAVE_ADMIN	Enable regular replication control. Level: Global.
RESOURCE_GROUP_ADMIN	Enable resource group administration. Level: Global.
RESOURCE_GROUP_USER	Enable resource group administration. Level: Global.
ROLE_ADMIN	Enable roles to be granted or revoked, use of WITH ADMIN OPTION. Level: Global.
SESSION_VARIABLES_ADMIN	Enable setting restricted session system variables. Level: Global.
SET_USER_ID	Enable setting non-self DEFINER values. Level: Global.
SHOW_ROUTINE	Enable access to stored routine definitions. Level: Global.
SKIP_QUERY_REWRITE	Do not rewrite queries executed by this user. Level: Global.
SYSTEM_USER	Designate account as system account. Level: Global.
SYSTEM_VARIABLES_ADMIN	Enable modifying or persisting global system variables. Level: Global.

Privilege	Meaning and Grantable Levels
TABLE_ENCRYPTION_ADMIN	Enable overriding default encryption settings. Level: Global.
TP_CONNECTION_ADMIN	Enable thread pool connection administration. Level: Global.
VERSION_TOKEN_ADMIN	Enable use of Version Tokens functions. Level: Global.
XA_RECOVER_ADMIN	Enable XA RECOVER execution. Level: Global.

A trigger is associated with a table. To create or drop a trigger, you must have the TRIGGER privilege for the table, not the trigger.

In GRANT statements, the ALL [PRIVILEGES] or PROXY privilege must be named by itself and cannot be specified along with other privileges. ALL [PRIVILEGES] stands for all privileges available for the level at which privileges are to be granted except for the GRANT OPTION and PROXY privileges.

MySQL account information is stored in the tables of the mysql system schema. For additional details, consult Section 6.2, “Access Control and Account Management”, which discusses the mysql system schema and the access control system extensively.

If the grant tables hold privilege rows that contain mixed-case database or table names and the lower_case_table_names system variable is set to a nonzero value, REVOKE cannot be used to revoke these privileges. It is necessary in such cases to manipulate the grant tables directly. (GRANT does not create such rows when lower_case_table_names is set, but such rows might have been created prior to setting that variable. The lower_case_table_names setting can only be configured at server startup.)

Privileges can be granted at several levels, depending on the syntax used for the ON clause. For REVOKE, the same ON syntax specifies which privileges to remove.

For the global, database, table, and routine levels, GRANT ALL assigns only the privileges that exist at the level you are granting. For example, GRANT ALL ON *db_name*.* is a database-level statement, so it does not grant any global-only privileges such as FILE. Granting ALL does not assign the GRANT OPTION or PROXY privilege.

The *object_type* clause, if present, should be specified as TABLE, FUNCTION, or PROCEDURE when the following object is a table, a stored function, or a stored procedure.

The privileges that a user holds for a database, table, column, or routine are formed additively as the logical OR of the account privileges at each of the privilege levels, including the global level. It is not possible to deny a privilege granted at a higher level by absence of that privilege at a lower level. For example, this statement grants the SELECT and INSERT privileges globally:

```
GRANT SELECT, INSERT ON *.* TO u1;
```

The globally granted privileges apply to all databases, tables, and columns, even though not granted at any of those lower levels.

As of MySQL 8.0.16, it is possible to explicitly deny a privilege granted at the global level by revoking it for particular databases, if the partial_revokes system variable is enabled:

```
GRANT SELECT, INSERT, UPDATE ON *.* TO u1;
REVOKE INSERT, UPDATE ON db1.* FROM u1;
```

The result of the preceding statements is that SELECT applies globally to all tables, whereas INSERT and UPDATE apply globally except to tables in db1. Account access to db1 is read only.

Details of the privilege-checking procedure are presented in Section 6.2.7, “Access Control, Stage 2: Request Verification”.

If you are using table, column, or routine privileges for even one user, the server examines table, column, and routine privileges for all users and this slows down MySQL a bit. Similarly, if you limit the number of queries, updates, or connections for any users, the server must monitor these values.

MySQL enables you to grant privileges on databases or tables that do not exist. For tables, the privileges to be granted must include the `CREATE` privilege. *This behavior is by design*, and is intended to enable the database administrator to prepare user accounts and privileges for databases or tables that are to be created at a later time.

Important

MySQL does not automatically revoke any privileges when you drop a database or table. However, if you drop a routine, any routine-level privileges granted for that routine are revoked.

Global Privileges

Global privileges are administrative or apply to all databases on a given server. To assign global privileges, use `ON *.*` syntax:

```
GRANT ALL ON *.* TO 'someuser'@'somehost';
GRANT SELECT, INSERT ON *.* TO 'someuser'@'somehost';
```

The `CREATE TABLESPACE`, `CREATE USER`, `FILE`, `PROCESS`, `RELOAD`, `REPLICATION CLIENT`, `REPLICATION SLAVE`, `SHOW DATABASES`, `SHUTDOWN`, and `SUPER` static privileges are administrative and can only be granted globally.

Dynamic privileges are all global and can only be granted globally.

Other privileges can be granted globally or at more specific levels.

The effect of `GRANT OPTION` granted at the global level differs for static and dynamic privileges:

- `GRANT OPTION` granted for any static global privilege applies to all static global privileges.
- `GRANT OPTION` granted for any dynamic privilege applies only to that dynamic privilege.

`GRANT ALL` at the global level grants all static global privileges and all currently registered dynamic privileges. A dynamic privilege registered subsequent to execution of the `GRANT` statement is not granted retroactively to any account.

MySQL stores global privileges in the `mysql.user` system table.

Database Privileges

Database privileges apply to all objects in a given database. To assign database-level privileges, use `ON *db_name*.*` syntax:

```
GRANT ALL ON mydb.* TO 'someuser'@'somehost';
GRANT SELECT, INSERT ON mydb.* TO 'someuser'@'somehost';
```

If you use `ON *` syntax (rather than `ON *.*`), privileges are assigned at the database level for the default database. An error occurs if there is no default database.

The `CREATE`, `DROP`, `EVENT`, `GRANT OPTION`, `LOCK TABLES`, and `REFERENCES` privileges can be specified at the database level. Table or routine privileges also can be specified at the database level, in which case they apply to all tables or routines in the database.

MySQL stores database privileges in the `mysql.db` system table.

Table Privileges

Table privileges apply to all columns in a given table. To assign table-level privileges, use `ON *db_name.tbl_name*` syntax:

```
GRANT ALL ON mydb.mytbl TO 'someuser'@'somehost';
GRANT SELECT, INSERT ON mydb.mytbl TO 'someuser'@'somehost';
```

If you specify *`tbl_name`* rather than *`db_name.tbl_name`*, the statement applies to *`tbl_name`* in the default database. An error occurs if there is no default database.

The permissible *`priv_type`* values at the table level are `ALTER`, `CREATE VIEW`, `CREATE`, `DELETE`, `DROP`, `GRANT OPTION`, `INDEX`, `INSERT`, `REFERENCES`, `SELECT`, `SHOW VIEW`, `TRIGGER`, and `UPDATE`.

Table-level privileges apply to base tables and views. They do not apply to tables created with `CREATE TEMPORARY TABLE`, even if the table names match. For information about `TEMPORARY` table privileges, see Section 13.1.20.2, “CREATE TEMPORARY TABLE Statement”.

MySQL stores table privileges in the `mysql.tables_priv` system table.

Column Privileges

Column privileges apply to single columns in a given table. Each privilege to be granted at the column level must be followed by the column or columns, enclosed within parentheses.

```
GRANT SELECT (col1), INSERT (col1, col2) ON mydb.mytbl TO 'someuser'@'somehost';
```

The permissible *`priv_type`* values for a column (that is, when you use a *`column_list`* clause) are `INSERT`, `REFERENCES`, `SELECT`, and `UPDATE`.

MySQL stores column privileges in the `mysql.columns_priv` system table.

Stored Routine Privileges

The `ALTER ROUTINE`, `CREATE ROUTINE`, `EXECUTE`, and `GRANT OPTION` privileges apply to stored routines (procedures and functions). They can be granted at the global and database levels. Except for `CREATE ROUTINE`, these privileges can be granted at the routine level for individual routines.

```
GRANT CREATE ROUTINE ON mydb.* TO 'someuser'@'somehost';
GRANT EXECUTE ON PROCEDURE mydb.myproc TO 'someuser'@'somehost';
```

The permissible *`priv_type`* values at the routine level are `ALTER ROUTINE`, `EXECUTE`, and `GRANT OPTION`. `CREATE ROUTINE` is not a routine-level privilege because you must have the privilege at the global or database level to create a routine in the first place.

MySQL stores routine-level privileges in the `mysql.procs_priv` system table.

Proxy User Privileges

The `PROXY` privilege enables one user to be a proxy for another. The proxy user impersonates or takes the identity of the proxied user; that is, it assumes the privileges of the proxied user.

```
GRANT PROXY ON 'localuser'@'localhost' TO 'externaluser'@'somehost';
```

When `PROXY` is granted, it must be the only privilege named in the `GRANT` statement, and the only permitted `WITH` option is `WITH GRANT OPTION`.

Proxying requires that the proxy user authenticate through a plugin that returns the name of the proxied user to the server when the proxy user connects, and that the proxy user have the `PROXY` privilege for the proxied user. For details and examples, see Section 6.2.19, “Proxy Users”.

MySQL stores proxy privileges in the `mysql.proxies_priv` system table.

Granting Roles

`GRANT` syntax without an `ON` clause grants roles rather than individual privileges. A role is a named collection of privileges; see Section 6.2.10, “Using Roles”. For example:

```
GRANT 'role1', 'role2' TO 'user1'@'localhost', 'user2'@'localhost';
```

Each role to be granted must exist, as well as each user account or role to which it is to be granted. As of MySQL 8.0.16, roles cannot be granted to anonymous users.

Granting a role does not automatically cause the role to be active. For information about role activation and inactivation, see Activating Roles.

These privileges are required to grant roles:

- If you have the `ROLE_ADMIN` privilege (or the deprecated `SUPER` privilege), you can grant or revoke any role to users or roles.
- If you were granted a role with a `GRANT` statement that includes the `WITH ADMIN OPTION` clause, you become able to grant that role to other users or roles, or revoke it from other users or roles, as long as the role is active at such time as you subsequently grant or revoke it. This includes the ability to use `WITH ADMIN OPTION` itself.
- To grant a role that has the `SYSTEM_USER` privilege, you must have the `SYSTEM_USER` privilege.

It is possible to create circular references with `GRANT`. For example:

```
CREATE USER 'u1', 'u2';
CREATE ROLE 'r1', 'r2';

GRANT 'u1' TO 'u1';   -- simple loop: u1 => u1
GRANT 'r1' TO 'r1';   -- simple loop: r1 => r1

GRANT 'r2' TO 'u2';
GRANT 'u2' TO 'r2';   -- mixed user/role loop: u2 => r2 => u2
```

Circular grant references are permitted but add no new privileges or roles to the grantee because a user or role already has its privileges and roles.

The `AS` Clause and Privilege Restrictions

As of MySQL 8.0.16, `GRANT` has an `AS *user* [WITH ROLE]` clause that specifies additional information about the privilege context to use for statement execution. This syntax is visible at the SQL level, although its primary purpose is to enable uniform replication across all nodes of grantor privilege restrictions imposed by partial revokes, by causing those restrictions to appear in the binary log. For information about partial revokes, see Section 6.2.12, “Privilege Restriction Using Partial Revokes”.

When the `AS *user*` clause is specified, statement execution takes into account any privilege restrictions associated with the named user, including all roles specified by `WITH ROLE`, if present. The result is that the privileges actually granted by the statement may be reduced relative to those specified.

These conditions apply to the `AS *user*` clause:

- `AS` has an effect only when the named *`user`* has privilege restrictions (which implies that the `partial_revokes` system variable is enabled).
- If `WITH ROLE` is given, all roles named must be granted to the named *`user`*.
- The named *`user`* should be a MySQL account specified as `'*user_name*'@'*host_name*'`, `CURRENT_USER`, or `CURRENT_USER()`. The current user may be named together with `WITH ROLE` for the case that the executing user wants `GRANT` to execute with a set of roles applied that may differ from the roles active within the current session.
- `AS` cannot be used to gain privileges not possessed by the user who executes the `GRANT` statement. The executing user must have at least the privileges to be granted, but the `AS` clause can only restrict the privileges granted, not escalate them.
- With respect to the privileges to be granted, `AS` cannot specify a user/role combination that has more privileges (fewer restrictions) than the user who executes the `GRANT` statement. The `AS` user/role

combination is permitted to have more privileges than the executing user, but only if the statement does not grant those additional privileges.

- AS is supported only for granting global privileges (ON *.*).
- AS is not supported for PROXY grants.

The following example illustrates the effect of the AS clause. Create a user u1 that has some global privileges, as well as restrictions on those privileges:

```
CREATE USER u1;
GRANT SELECT, INSERT, UPDATE, DELETE ON *.* TO u1;
REVOKE INSERT, UPDATE ON schema1.* FROM u1;
REVOKE SELECT ON schema2.* FROM u1;
```

Also create a role r1 that lifts some of the privilege restrictions and grant the role to u1:

```
CREATE ROLE r1;
GRANT INSERT ON schema1.* TO r1;
GRANT SELECT ON schema2.* TO r1;
GRANT r1 TO u1;
```

Now, using an account that has no privilege restrictions of its own, grant to multiple users the same set of global privileges, but each with different restrictions imposed by the AS clause, and check which privileges are actually granted.

- The GRANT statement here has no AS clause, so the privileges granted are exactly those specified:

```
mysql> CREATE USER u2;
mysql> GRANT SELECT, INSERT, UPDATE ON *.* TO u2;
mysql> SHOW GRANTS FOR u2;
+-------------------------------------------------+
| Grants for u2@%                                 |
+-------------------------------------------------+
| GRANT SELECT, INSERT, UPDATE ON *.* TO `u2`@`%` |
+-------------------------------------------------+
```

- The GRANT statement here has an AS clause, so the privileges granted are those specified but with the restrictions from u1 applied:

```
mysql> CREATE USER u3;
mysql> GRANT SELECT, INSERT, UPDATE ON *.* TO u3 AS u1;
mysql> SHOW GRANTS FOR u3;
+-----------------------------------------------------+
| Grants for u3@%                                     |
+-----------------------------------------------------+
| GRANT SELECT, INSERT, UPDATE ON *.* TO `u3`@`%`     |
| REVOKE INSERT, UPDATE ON `schema1`.* FROM `u3`@`%`  |
| REVOKE SELECT ON `schema2`.* FROM `u3`@`%`          |
+-----------------------------------------------------+
```

  As mentioned previously, the AS clause can only add privilege restrictions; it cannot escalate privileges. Thus, although u1 has the DELETE privilege, that is not included in the privileges granted because the statement does not specify granting DELETE.

- The AS clause for the GRANT statement here makes the role r1 active for u1. That role lifts some of the restrictions on u1. Consequently, the privileges granted have some restrictions, but not so many as for the previous GRANT statement:

```
mysql> CREATE USER u4;
mysql> GRANT SELECT, INSERT, UPDATE ON *.* TO u4 AS u1 WITH ROLE r1;
mysql> SHOW GRANTS FOR u4;
+-------------------------------------------------+
| Grants for u4@%                                 |
+-------------------------------------------------+
| GRANT SELECT, INSERT, UPDATE ON *.* TO `u4`@`%` |
| REVOKE UPDATE ON `schema1`.* FROM `u4`@`%`      |
+-------------------------------------------------+
```

If a GRANT statement includes an AS *user* clause, privilege restrictions on the user who executes the statement are ignored (rather than applied as they would be in the absence of an AS clause).

Other Account Characteristics

The optional WITH clause is used to enable a user to grant privileges to other users. The WITH GRANT OPTION clause gives the user the ability to give to other users any privileges the user has at the specified privilege level.

To grant the GRANT OPTION privilege to an account without otherwise changing its privileges, do this:

```
GRANT USAGE ON *.* TO 'someuser'@'somehost' WITH GRANT OPTION;
```

Be careful to whom you give the GRANT OPTION privilege because two users with different privileges may be able to combine privileges!

You cannot grant another user a privilege which you yourself do not have; the GRANT OPTION privilege enables you to assign only those privileges which you yourself possess.

Be aware that when you grant a user the GRANT OPTION privilege at a particular privilege level, any privileges the user possesses (or may be given in the future) at that level can also be granted by that user to other users. Suppose that you grant a user the INSERT privilege on a database. If you then grant the SELECT privilege on the database and specify WITH GRANT OPTION, that user can give to other users not only the SELECT privilege, but also INSERT. If you then grant the UPDATE privilege to the user on the database, the user can grant INSERT, SELECT, and UPDATE.

For a nonadministrative user, you should not grant the ALTER privilege globally or for the mysql system schema. If you do that, the user can try to subvert the privilege system by renaming tables!

For additional information about security risks associated with particular privileges, see Section 6.2.2, “Privileges Provided by MySQL”.

MySQL and Standard SQL Versions of GRANT

The biggest differences between the MySQL and standard SQL versions of GRANT are:

- MySQL associates privileges with the combination of a host name and user name and not with only a user name.
- Standard SQL does not have global or database-level privileges, nor does it support all the privilege types that MySQL supports.
- MySQL does not support the standard SQL UNDER privilege.
- Standard SQL privileges are structured in a hierarchical manner. If you remove a user, all privileges the user has been granted are revoked. This is also true in MySQL if you use DROP USER. See Section 13.7.1.5, “DROP USER Statement”.
- In standard SQL, when you drop a table, all privileges for the table are revoked. In standard SQL, when you revoke a privilege, all privileges that were granted based on that privilege are also revoked. In MySQL, privileges can be dropped with DROP USER or REVOKE statements.
- In MySQL, it is possible to have the INSERT privilege for only some of the columns in a table. In this case, you can still execute INSERT statements on the table, provided that you insert values only for those columns for which you have the INSERT privilege. The omitted columns are set to their implicit default values if strict SQL mode is not enabled. In strict mode, the statement is rejected if any of the omitted columns have no default value. (Standard SQL requires you to have the INSERT privilege on all columns.) For information about strict SQL mode and implicit default values, see Section 5.1.11, “Server SQL Modes”, and Section 11.6, “Data Type Default Values”.

13.7.1.7 RENAME USER Statement

```
RENAME USER old_user TO new_user
    [, old_user TO new_user] ...
```

The RENAME USER statement renames existing MySQL accounts. An error occurs for old accounts that do not exist or new accounts that already exist.

To use RENAME USER, you must have the global CREATE USER privilege, or the UPDATE privilege for the mysql system schema. When the read_only system variable is enabled, RENAME USER additionally requires the CONNECTION_ADMIN privilege (or the deprecated SUPER privilege).

As of MySQL 8.0.22, RENAME USER fails with an error if any account to be renamed is named as the DEFINER attribute for any stored object. (That is, the statement fails if renaming an account would cause a stored object to become orphaned.) To perform the operation anyway, you must have the SET_USER_ID privilege; in this case, the statement succeeds with a warning rather than failing with an error. For additional information, including how to identify which objects name a given account as the DEFINER attribute, see Orphan Stored Objects.

Each account name uses the format described in Section 6.2.4, “Specifying Account Names”. For example:

```
RENAME USER 'jeffrey'@'localhost' TO 'jeff'@'127.0.0.1';
```

The host name part of the account name, if omitted, defaults to '%'.

RENAME USER causes the privileges held by the old user to be those held by the new user. However, RENAME USER does not automatically drop or invalidate databases or objects within them that the old user created. This includes stored programs or views for which the DEFINER attribute names the old user. Attempts to access such objects may produce an error if they execute in definer security context. (For information about security context, see Section 25.6, “Stored Object Access Control”.)

The privilege changes take effect as indicated in Section 6.2.13, “When Privilege Changes Take Effect”.

13.7.1.8 REVOKE Statement

```
REVOKE [IF EXISTS]
    priv_type [(column_list)]
      [, priv_type [(column_list)]] ...
    ON [object_type] priv_level
    FROM user_or_role [, user_or_role] ...
    [IGNORE UNKNOWN USER]

REVOKE [IF EXISTS] ALL [PRIVILEGES], GRANT OPTION
    FROM user_or_role [, user_or_role] ...
    [IGNORE UNKNOWN USER]

REVOKE [IF EXISTS] PROXY ON user_or_role
    FROM user_or_role [, user_or_role] ...
    [IGNORE UNKNOWN USER]

REVOKE [IF EXISTS] role [, role ] ...
    FROM user_or_role [, user_or_role ] ...
    [IGNORE UNKNOWN USER]

user_or_role: {
    user (see Section 6.2.4, “Specifying Account Names”)
  | role (see Section 6.2.5, “Specifying Role Names”
}
```

The REVOKE statement enables system administrators to revoke privileges and roles, which can be revoked from user accounts and roles.

For details on the levels at which privileges exist, the permissible *priv_type*, *priv_level*, and *object_type* values, and the syntax for specifying users and passwords, see Section 13.7.1.6, “GRANT Statement”.

For information about roles, see Section 6.2.10, “Using Roles”.

When the `read_only` system variable is enabled, `REVOKE` requires the `CONNECTION_ADMIN` or privilege (or the deprecated `SUPER` privilege), in addition to any other required privileges described in the following discussion.

Beginning with MySQL 8.0.30, all the forms shown for `REVOKE` support an `IF EXISTS` option as well as an `IGNORE UNKNOWN USER` option. With neither of these modifications, `REVOKE` either succeeds for all named users and roles, or rolls back and has no effect if any error occurs; the statement is written to the binary log only if it succeeds for all named users and roles. The precise effects of `IF EXISTS` and `IGNORE UNKNOWN USER` are discussed later in this section.

Each account name uses the format described in Section 6.2.4, “Specifying Account Names”. Each role name uses the format described in Section 6.2.5, “Specifying Role Names”. For example:

```
REVOKE INSERT ON *.* FROM 'jeffrey'@'localhost';
REVOKE 'role1', 'role2' FROM 'user1'@'localhost', 'user2'@'localhost';
REVOKE SELECT ON world.* FROM 'role3';
```

The host name part of the account or role name, if omitted, defaults to `'%'`.

To use the first `REVOKE` syntax, you must have the `GRANT OPTION` privilege, and you must have the privileges that you are revoking.

To revoke all privileges, use the second syntax, which drops all global, database, table, column, and routine privileges for the named users or roles:

```
REVOKE ALL PRIVILEGES, GRANT OPTION
  FROM user_or_role [, user_or_role] ...
```

`REVOKE ALL PRIVILEGES, GRANT OPTION` does not revoke any roles.

To use this `REVOKE` syntax, you must have the global `CREATE USER` privilege, or the `UPDATE` privilege for the `mysql` system schema.

The syntax for which the `REVOKE` keyword is followed by one or more role names takes a `FROM` clause indicating one or more users or roles from which to revoke the roles.

The `IF EXISTS` and `IGNORE UNKNOWN USER` options (MySQL 8.0.30 and later) have the effects listed here:

- `IF EXISTS` means that, if the target user or role exists but no such privilege or role is found assigned to the target for any reason, a warning is raised, instead of an error; if no privilege or role named by the statement is assigned to the target, the statement has no (other) effect. Otherwise, `REVOKE` executes normally; if the user does not exist, the statement raises an error.

  Example: Given table `t1` in database `test`, we execute the following statements, with the results shown.

  ```
  mysql> CREATE USER jerry@localhost;
  Query OK, 0 rows affected (0.01 sec)

  mysql> REVOKE SELECT ON test.t1 FROM jerry@localhost;
  ERROR 1147 (42000): There is no such grant defined for user 'jerry' on host
  'localhost' on table 't1'
  mysql> REVOKE IF EXISTS SELECT ON test.t1 FROM jerry@localhost;
  Query OK, 0 rows affected, 1 warning (0.00 sec)

  mysql> SHOW WARNINGS\G
  *************************** 1. row ***************************
    Level: Warning
     Code: 1147
  Message: There is no such grant defined for user 'jerry' on host 'localhost' on
  table 't1'
  ```

```
1 row in set (0.00 sec)
```

`IF EXISTS` causes an error to be demoted to a warning even if the privilege or role named does not exist, or the statement attempts to assign it at the wrong level.

- If the `REVOKE` statement includes `IGNORE UNKNOWN USER`, the statement raises a warning for any target user or role named in the statement but not found; if no target named by the statement exists, `REVOKE` succeeds but has no actual effect. Otherwise, the statement executes as usual, and attempting to revoke a privilege not assigned to the target for whatever reason raises an error, as expected.

  Example (continuing from the previous example):

```
mysql> DROP USER IF EXISTS jerry@localhost;
Query OK, 0 rows affected (0.01 sec)

mysql> REVOKE SELECT ON test.t1 FROM jerry@localhost;
ERROR 1147 (42000): There is no such grant defined for user 'jerry' on host
'localhost' on table 't1'
mysql> REVOKE SELECT ON test.t1 FROM jerry@localhost IGNORE UNKNOWN USER;
Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
   Code: 3162
Message: Authorization ID jerry does not exist.
1 row in set (0.00 sec)
```

- The combination of `IF EXISTS` and `IGNORE UNKNOWN USER` means that `REVOKE` never raises an error for an unknown target user or role or for an unassigned or unavailable privilege, and the statement as whole in such cases succeeds; roles or privileges are removed from existing target users or roles whenever possible, and any revocation which is not possible raises a warning and executes as a `NOOP`.

  Example (again continuing from example in the previous item):

```
# No such user, no such role
mysql> DROP ROLE IF EXISTS Bogus;
Query OK, 0 rows affected, 1 warning (0.02 sec)

mysql> SHOW WARNINGS;
+-------+------+-----------------------------------------------+
| Level | Code | Message                                       |
+-------+------+-----------------------------------------------+
| Note  | 3162 | Authorization ID 'Bogus'@'%' does not exist. |
+-------+------+-----------------------------------------------+
1 row in set (0.00 sec)

# This statement attempts to revoke a nonexistent role from a nonexistent user
mysql> REVOKE Bogus ON test FROM jerry@localhost;
ERROR 3619 (HY000): Illegal privilege level specified for test

# The same, with IF EXISTS
mysql> REVOKE IF EXISTS Bogus ON test FROM jerry@localhost;
ERROR 1147 (42000): There is no such grant defined for user 'jerry' on host
'localhost' on table 'test'

# The same, with IGNORE UNKNOWN USER
mysql> REVOKE Bogus ON test FROM jerry@localhost IGNORE UNKNOWN USER;
ERROR 3619 (HY000): Illegal privilege level specified for test

# The same, with both options
mysql> REVOKE IF EXISTS Bogus ON test FROM jerry@localhost IGNORE UNKNOWN USER;
Query OK, 0 rows affected, 2 warnings (0.01 sec)

mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------------+
| Level   | Code | Message                                       |
```

```
+---------+------+-----------------------------------------+
| Warning | 3619 | Illegal privilege level specified for test |
| Warning | 3162 | Authorization ID jerry does not exist.     |
+---------+------+-----------------------------------------+
2 rows in set (0.00 sec)
```

Roles named in the `mandatory_roles` system variable value cannot be revoked. When `IF EXISTS` and `IGNORE UNKNOWN USER` are used together in a statement that tries to remove a mandatory privilege, the error normally raised by attempting to do this is demoted to a warning; the statement executes successfully, but does not make any changes.

A revoked role immediately affects any user account from which it was revoked, such that within any current session for the account, its privileges are adjusted for the next statement executed.

Revoking a role revokes the role itself, not the privileges that it represents. Suppose that an account is granted a role that includes a given privilege, and is also granted the privilege explicitly or another role that includes the privilege. In this case, the account still possesses that privilege if the first role is revoked. For example, if an account is granted two roles that each include `SELECT`, the account still can select after either role is revoked.

`REVOKE ALL ON *.*` (at the global level) revokes all granted static global privileges and all granted dynamic privileges.

A revoked privilege that is granted but not known to the server is revoked with a warning. This situation can occur for dynamic privileges. For example, a dynamic privilege can be granted while the component that registers it is installed, but if that component is subsequently uninstalled, the privilege becomes unregistered, although accounts that possess the privilege still possess it and it can be revoked from them.

`REVOKE` removes privileges, but does not remove rows from the `mysql.user` system table. To remove a user account entirely, use `DROP USER`. See Section 13.7.1.5, “DROP USER Statement”.

If the grant tables hold privilege rows that contain mixed-case database or table names and the `lower_case_table_names` system variable is set to a nonzero value, `REVOKE` cannot be used to revoke these privileges. It is necessary in such cases to manipulate the grant tables directly. (`GRANT` does not create such rows when `lower_case_table_names` is set, but such rows might have been created prior to setting the variable. The `lower_case_table_names` setting can only be configured when initializing the server.)

When successfully executed from the `mysql` program, `REVOKE` responds with `Query OK, 0 rows affected`. To determine what privileges remain after the operation, use `SHOW GRANTS`. See Section 13.7.7.21, “SHOW GRANTS Statement”.

13.7.1.9 SET DEFAULT ROLE Statement

```
SET DEFAULT ROLE
    {NONE | ALL | role [, role ] ...}
    TO user [, user ] ...
```

For each *`user`* named immediately after the `TO` keyword, this statement defines which roles become active when the user connects to the server and authenticates, or when the user executes the `SET ROLE DEFAULT` statement during a session.

`SET DEFAULT ROLE` is alternative syntax for `ALTER USER ... DEFAULT ROLE` (see Section 13.7.1.1, “ALTER USER Statement”). However, `ALTER USER` can set the default for only a single user, whereas `SET DEFAULT ROLE` can set the default for multiple users. On the other hand, you can specify `CURRENT_USER` as the user name for the `ALTER USER` statement, whereas you cannot for `SET DEFAULT ROLE`.

`SET DEFAULT ROLE` requires these privileges:

- Setting the default roles for another user requires the global `CREATE USER` privilege, or the `UPDATE` privilege for the `mysql.default_roles` system table.

- Setting the default roles for yourself requires no special privileges, as long as the roles you want as the default have been granted to you.

Each role name uses the format described in Section 6.2.5, “Specifying Role Names”. For example:

```
SET DEFAULT ROLE 'admin', 'developer' TO 'joe'@'10.0.0.1';
```

The host name part of the role name, if omitted, defaults to `'%'`.

The clause following the `DEFAULT ROLE` keywords permits these values:

- `NONE`: Set the default to `NONE` (no roles).
- `ALL`: Set the default to all roles granted to the account.
- *`role [, role ] ...`*: Set the default to the named roles, which must exist and be granted to the account at the time `SET DEFAULT ROLE` is executed.

Note

`SET DEFAULT ROLE` and `SET ROLE DEFAULT` are different statements:

- `SET DEFAULT ROLE` defines which account roles to activate by default within account sessions.
- `SET ROLE DEFAULT` sets the active roles within the current session to the current account default roles.

For role usage examples, see Section 6.2.10, “Using Roles”.

13.7.1.10 SET PASSWORD Statement

```
SET PASSWORD [FOR user] auth_option
    [REPLACE 'current_auth_string']
    [RETAIN CURRENT PASSWORD]

auth_option: {
    = 'auth_string'
  | TO RANDOM
}
```

The `SET PASSWORD` statement assigns a password to a MySQL user account. The password may be either explicitly specified in the statement or randomly generated by MySQL. The statement may also include a password-verification clause that specifies the account current password to be replaced, and a clause that manages whether an account has a secondary password. *`'auth_string'`* and *`'current_auth_string'`* each represent a cleartext (unencrypted) password.

Note

Rather than using `SET PASSWORD` to assign passwords, `ALTER USER` is the preferred statement for account alterations, including assigning passwords. For example:

```
ALTER USER user IDENTIFIED BY 'auth_string';
```

Note

Clauses for random password generation, password verification, and secondary passwords apply only to accounts that use an authentication plugin that stores credentials internally to MySQL. For accounts that use a plugin that performs authentication against a credentials system that is external to MySQL, password management must be handled externally against that system as well. For more information about internal credentials storage, see Section 6.2.15, “Password Management”.

The `REPLACE 'current_auth_string'` clause performs password verification and is available as of MySQL 8.0.13. If given:

- `REPLACE` specifies the account current password to be replaced, as a cleartext (unencrypted) string.
- The clause must be given if password changes for the account are required to specify the current password, as verification that the user attempting to make the change actually knows the current password.
- The clause is optional if password changes for the account may but need not specify the current password.
- The statement fails if the clause is given but does not match the current password, even if the clause is optional.
- `REPLACE` can be specified only when changing the account password for the current user.

For more information about password verification by specifying the current password, see Section 6.2.15, “Password Management”.

The `RETAIN CURRENT PASSWORD` clause implements dual-password capability and is available as of MySQL 8.0.14. If given:

- `RETAIN CURRENT PASSWORD` retains an account current password as its secondary password, replacing any existing secondary password. The new password becomes the primary password, but clients can use the account to connect to the server using either the primary or secondary password. (Exception: If the new password specified by the `SET PASSWORD` statement is empty, the secondary password becomes empty as well, even if `RETAIN CURRENT PASSWORD` is given.)
- If you specify `RETAIN CURRENT PASSWORD` for an account that has an empty primary password, the statement fails.
- If an account has a secondary password and you change its primary password without specifying `RETAIN CURRENT PASSWORD`, the secondary password remains unchanged.

For more information about use of dual passwords, see Section 6.2.15, “Password Management”.

`SET PASSWORD` permits these *`auth_option`* syntaxes:

- `= 'auth_string'`

  Assigns the account the given literal password.

- `TO RANDOM`

  Assigns the account a password randomly generated by MySQL. The statement also returns the cleartext password in a result set to make it available to the user or application executing the statement.

  For details about the result set and characteristics of randomly generated passwords, see Random Password Generation.

  Random password generation is available as of MySQL 8.0.18.

Important

Under some circumstances, `SET PASSWORD` may be recorded in server logs or on the client side in a history file such as `~/.mysql_history`, which means that cleartext passwords may be read by anyone having read access to that information. For information about the conditions under which this occurs for the server logs and how to control it, see Section 6.1.2.3, “Passwords and Logging”.

> For similar information about client-side logging, see Section 4.5.1.3, “mysql Client Logging”.

`SET PASSWORD` can be used with or without a `FOR` clause that explicitly names a user account:

- With a `FOR user` clause, the statement sets the password for the named account, which must exist:

```
SET PASSWORD FOR 'jeffrey'@'localhost' = 'auth_string';
```

- With no `FOR user` clause, the statement sets the password for the current user:

```
SET PASSWORD = 'auth_string';
```

  Any client who connects to the server using a nonanonymous account can change the password for that account. (In particular, you can change your own password.) To see which account the server authenticated you as, invoke the `CURRENT_USER()` function:

```
SELECT CURRENT_USER();
```

If a `FOR user` clause is given, the account name uses the format described in Section 6.2.4, “Specifying Account Names”. For example:

```
SET PASSWORD FOR 'bob'@'%.example.org' = 'auth_string';
```

The host name part of the account name, if omitted, defaults to `'%'`.

`SET PASSWORD` interprets the string as a cleartext string, passes it to the authentication plugin associated with the account, and stores the result returned by the plugin in the account row in the `mysql.user` system table. (The plugin is given the opportunity to hash the value into the encryption format it expects. The plugin may use the value as specified, in which case no hashing occurs.)

Setting the password for a named account (with a `FOR` clause) requires the `UPDATE` privilege for the `mysql` system schema. Setting the password for yourself (for a nonanonymous account with no `FOR` clause) requires no special privileges.

Statements that modify secondary passwords require these privileges:

- The `APPLICATION_PASSWORD_ADMIN` privilege is required to use the `RETAIN CURRENT PASSWORD` clause for `SET PASSWORD` statements that apply to your own account. The privilege is required to manipulate your own secondary password because most users require only one password.
- If an account is to be permitted to manipulate secondary passwords for all accounts, it should be granted the `CREATE USER` privilege rather than `APPLICATION_PASSWORD_ADMIN`.

When the `read_only` system variable is enabled, `SET PASSWORD` requires the `CONNECTION_ADMIN` privilege (or the deprecated `SUPER` privilege), in addition to any other required privileges.

For additional information about setting passwords and authentication plugins, see Section 6.2.14, “Assigning Account Passwords”, and Section 6.2.17, “Pluggable Authentication”.

13.7.1.11 SET ROLE Statement

```
SET ROLE {
    DEFAULT
  | NONE
  | ALL
  | ALL EXCEPT role [, role ] ...
  | role [, role ] ...
}
```

`SET ROLE` modifies the current user's effective privileges within the current session by specifying which of its granted roles are active. Granted roles include those granted explicitly to the user and those named in the `mandatory_roles` system variable value.

Examples:

```
SET ROLE DEFAULT;
SET ROLE 'role1', 'role2';
SET ROLE ALL;
SET ROLE ALL EXCEPT 'role1', 'role2';
```

Each role name uses the format described in Section 6.2.5, “Specifying Role Names”. The host name part of the role name, if omitted, defaults to `'%'`.

Privileges that the user has been granted directly (rather than through roles) remain unaffected by changes to the active roles.

The statement permits these role specifiers:

- `DEFAULT`: Activate the account default roles. Default roles are those specified with `SET DEFAULT ROLE`.

  When a user connects to the server and authenticates successfully, the server determines which roles to activate as the default roles. If the `activate_all_roles_on_login` system variable is enabled, the server activates all granted roles. Otherwise, the server executes `SET ROLE DEFAULT` implicitly. The server activates only default roles that can be activated. The server writes warnings to its error log for default roles that cannot be activated, but the client receives no warnings.

  If a user executes `SET ROLE DEFAULT` during a session, an error occurs if any default role cannot be activated (for example, if it does not exist or is not granted to the user). In this case, the current active roles are not changed.

- `NONE`: Set the active roles to `NONE` (no active roles).

- `ALL`: Activate all roles granted to the account.

- `ALL EXCEPT` *`role [, role ] ...`*: Activate all roles granted to the account except those named. The named roles need not exist or be granted to the account.

- *`role [, role ] ...`*: Activate the named roles, which must be granted to the account.

Note

`SET DEFAULT ROLE` and `SET ROLE DEFAULT` are different statements:

- `SET DEFAULT ROLE` defines which account roles to activate by default within account sessions.
- `SET ROLE DEFAULT` sets the active roles within the current session to the current account default roles.

For role usage examples, see Section 6.2.10, “Using Roles”.

13.7.2 Resource Group Management Statements

MySQL supports creation and management of resource groups, and permits assigning threads running within the server to particular groups so that threads execute according to the resources available to the group. This section describes the SQL statements available for resource group management. For general discussion of the resource group capability, see Section 5.1.16, “Resource Groups”.

13.7.2.1 ALTER RESOURCE GROUP Statement

```
ALTER RESOURCE GROUP group_name
    [VCPU [=] vcpu_spec [, vcpu_spec] ...]
    [THREAD_PRIORITY [=] N]
    [ENABLE|DISABLE [FORCE]]
```

```
vcpu_spec: {N | M - N}
```

`ALTER RESOURCE GROUP` is used for resource group management (see Section 5.1.16, “Resource Groups”). This statement alters modifiable attributes of an existing resource group. It requires the `RESOURCE_GROUP_ADMIN` privilege.

`group_name` identifies which resource group to alter. If the group does not exist, an error occurs.

The attributes for CPU affinity, priority, and whether the group is enabled can be modified with `ALTER RESOURCE GROUP`. These attributes are specified the same way as described for `CREATE RESOURCE GROUP` (see Section 13.7.2.2, “CREATE RESOURCE GROUP Statement”). Only the attributes specified are altered. Unspecified attributes retain their current values.

The `FORCE` modifier is used with `DISABLE`. It determines statement behavior if the resource group has any threads assigned to it:

- If `FORCE` is not given, existing threads in the group continue to run until they terminate, but new threads cannot be assigned to the group.
- If `FORCE` is given, existing threads in the group are moved to their respective default group (system threads to `SYS_default`, user threads to `USR_default`).

The name and type attributes are set at group creation time and cannot be modified thereafter with `ALTER RESOURCE GROUP`.

Examples:

- Alter a group CPU affinity:

  ```
  ALTER RESOURCE GROUP rg1 VCPU = 0-63;
  ```

- Alter a group thread priority:

  ```
  ALTER RESOURCE GROUP rg2 THREAD_PRIORITY = 5;
  ```

- Disable a group, moving any threads assigned to it to the default groups:

  ```
  ALTER RESOURCE GROUP rg3 DISABLE FORCE;
  ```

Resource group management is local to the server on which it occurs. `ALTER RESOURCE GROUP` statements are not written to the binary log and are not replicated.

13.7.2.2 CREATE RESOURCE GROUP Statement

```
CREATE RESOURCE GROUP group_name
    TYPE = {SYSTEM|USER}
    [VCPU [=] vcpu_spec [, vcpu_spec] ...]
    [THREAD_PRIORITY [=] N]
    [ENABLE|DISABLE]

vcpu_spec: {N | M - N}
```

`CREATE RESOURCE GROUP` is used for resource group management (see Section 5.1.16, “Resource Groups”). This statement creates a new resource group and assigns its initial attribute values. It requires the `RESOURCE_GROUP_ADMIN` privilege.

`group_name` identifies which resource group to create. If the group already exists, an error occurs.

The `TYPE` attribute is required. It should be `SYSTEM` for a system resource group, `USER` for a user resource group. The group type affects permitted `THREAD_PRIORITY` values, as described later.

The `VCPU` attribute indicates the CPU affinity; that is, the set of virtual CPUs the group can use:

- If `VCPU` is not given, the resource group has no CPU affinity and can use all available CPUs.

- If VCPU is given, the attribute value is a list of comma-separated CPU numbers or ranges:
  - Each number must be an integer in the range from 0 to the number of CPUs − 1. For example, on a system with 64 CPUs, the number can range from 0 to 63.
  - A range is given in the form *M* − *N*, where *M* is less than or equal to *N* and both numbers are in the CPU range.
  - If a CPU number is an integer outside the permitted range or is not an integer, an error occurs.

Example VCPU specifiers (these are all equivalent):

```
VCPU = 0,1,2,3,9,10
VCPU = 0-3,9-10
VCPU = 9,10,0-3
VCPU = 0,10,1,9,3,2
```

The THREAD_PRIORITY attribute indicates the priority for threads assigned to the group:

- If THREAD_PRIORITY is not given, the default priority is 0.
- If THREAD_PRIORITY is given, the attribute value must be in the range from -20 (highest priority) to 19 (lowest priority). The priority for system resource groups must be in the range from -20 to 0. The priority for user resource groups must be in the range from 0 to 19. Use of different ranges for system and user groups ensures that user threads never have a higher priority than system threads.

ENABLE and DISABLE specify that the resource group is initially enabled or disabled. If neither is specified, the group is enabled by default. A disabled group cannot have threads assigned to it.

Examples:

- Create an enabled user group that has a single CPU and the lowest priority:

```
CREATE RESOURCE GROUP rg1
  TYPE = USER
  VCPU = 0
  THREAD_PRIORITY = 19;
```

- Create a disabled system group that has no CPU affinity (can use all CPUs) and the highest priority:

```
CREATE RESOURCE GROUP rg2
  TYPE = SYSTEM
  THREAD_PRIORITY = -20
  DISABLE;
```

Resource group management is local to the server on which it occurs. CREATE RESOURCE GROUP statements are not written to the binary log and are not replicated.

13.7.2.3 DROP RESOURCE GROUP Statement

```
DROP RESOURCE GROUP group_name [FORCE]
```

DROP RESOURCE GROUP is used for resource group management (see Section 5.1.16, “Resource Groups”). This statement drops a resource group. It requires the RESOURCE_GROUP_ADMIN privilege.

group_name identifies which resource group to drop. If the group does not exist, an error occurs.

The FORCE modifier determines statement behavior if the resource group has any threads assigned to it:

- If FORCE is not given and any threads are assigned to the group, an error occurs.
- If FORCE is given, existing threads in the group are moved to their respective default group (system threads to SYS_default, user threads to USR_default).

Examples:

- Drop a group, failing if the group contains any threads:

```
DROP RESOURCE GROUP rg1;
```

- Drop a group and move existing threads to the default groups:

```
DROP RESOURCE GROUP rg2 FORCE;
```

Resource group management is local to the server on which it occurs. `DROP RESOURCE GROUP` statements are not written to the binary log and are not replicated.

13.7.2.4 SET RESOURCE GROUP Statement

```
SET RESOURCE GROUP group_name
    [FOR thread_id [, thread_id] ...]
```

`SET RESOURCE GROUP` is used for resource group management (see Section 5.1.16, "Resource Groups"). This statement assigns threads to a resource group. It requires the `RESOURCE_GROUP_ADMIN` or `RESOURCE_GROUP_USER` privilege.

`group_name` identifies which resource group to be assigned. Any *`thread_id`* values indicate threads to assign to the group. Thread IDs can be determined from the Performance Schema `threads` table. If the resource group or any named thread ID does not exist, an error occurs.

With no `FOR` clause, the statement assigns the current thread for the session to the resource group.

With a `FOR` clause that names thread IDs, the statement assigns those threads to the resource group.

For attempts to assign a system thread to a user resource group or a user thread to a system resource group, a warning occurs.

Examples:

- Assign the current session thread to a group:

```
SET RESOURCE GROUP rg1;
```

- Assign the named threads to a group:

```
SET RESOURCE GROUP rg2 FOR 14, 78, 4;
```

Resource group management is local to the server on which it occurs. `SET RESOURCE GROUP` statements are not written to the binary log and are not replicated.

An alternative to `SET RESOURCE GROUP` is the `RESOURCE_GROUP` optimizer hint, which assigns individual statements to a resource group. See Section 8.9.3, "Optimizer Hints".

13.7.3 Table Maintenance Statements

13.7.3.1 ANALYZE TABLE Statement

```
ANALYZE [NO_WRITE_TO_BINLOG | LOCAL]
    TABLE tbl_name [, tbl_name] ...

ANALYZE [NO_WRITE_TO_BINLOG | LOCAL]
    TABLE tbl_name
    UPDATE HISTOGRAM ON col_name [, col_name] ...
        [WITH N BUCKETS]

ANALYZE [NO_WRITE_TO_BINLOG | LOCAL]
    TABLE tbl_name
    UPDATE HISTOGRAM ON col_name [USING DATA 'json_data']
```

```
ANALYZE [NO_WRITE_TO_BINLOG | LOCAL]
    TABLE tbl_name
    DROP HISTOGRAM ON col_name [, col_name] ...
```

`ANALYZE TABLE` generates table statistics:

- `ANALYZE TABLE` without either `HISTOGRAM` clause performs a key distribution analysis and stores the distribution for the named table or tables. For `MyISAM` tables, `ANALYZE TABLE` for key distribution analysis is equivalent to using `myisamchk --analyze`.
- `ANALYZE TABLE` with the `UPDATE HISTOGRAM` clause generates histogram statistics for the named table columns and stores them in the data dictionary. Only one table name is permitted for this syntax. MySQL 8.0.31 and later also supports setting the histogram of a single column to a user-defined JSON value.
- `ANALYZE TABLE` with the `DROP HISTOGRAM` clause removes histogram statistics for the named table columns from the data dictionary. Only one table name is permitted for this syntax.

This statement requires `SELECT` and `INSERT` privileges for the table.

`ANALYZE TABLE` works with `InnoDB`, `NDB`, and `MyISAM` tables. It does not work with views.

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`.

`ANALYZE TABLE` is supported for partitioned tables, and you can use `ALTER TABLE ... ANALYZE PARTITION` to analyze one or more partitions; for more information, see Section 13.1.9, “ALTER TABLE Statement”, and Section 24.3.4, “Maintenance of Partitions”.

During the analysis, the table is locked with a read lock for `InnoDB` and `MyISAM`.

`ANALYZE TABLE` removes the table from the table definition cache, which requires a flush lock. If there are long running statements or transactions still using the table, subsequent statements and transactions must wait for those operations to finish before the flush lock is released. Because `ANALYZE TABLE` itself typically finishes quickly, it may not be apparent that delayed transactions or statements involving the same table are due to the remaining flush lock.

By default, the server writes `ANALYZE TABLE` statements to the binary log so that they replicate to replicas. To suppress logging, specify the optional `NO_WRITE_TO_BINLOG` keyword or its alias `LOCAL`.

- ANALYZE TABLE Output
- Key Distribution Analysis
- Histogram Statistics Analysis
- Other Considerations

ANALYZE TABLE Output

`ANALYZE TABLE` returns a result set with the columns shown in the following table.

Column	Value
`Table`	The table name
`Op`	`analyze` or `histogram`
`Msg_type`	`status`, `error`, `info`, `note`, or `warning`
`Msg_text`	An informational message

Key Distribution Analysis

`ANALYZE TABLE` without either `HISTOGRAM` clause performs a key distribution analysis and stores the distribution for the table or tables. Any existing histogram statistics remain unaffected.

If the table has not changed since the last key distribution analysis, the table is not analyzed again.

MySQL uses the stored key distribution to decide the order in which tables should be joined for joins on something other than a constant. In addition, key distributions can be used when deciding which indexes to use for a specific table within a query.

To check the stored key distribution cardinality, use the `SHOW INDEX` statement or the `INFORMATION_SCHEMA STATISTICS` table. See Section 13.7.7.22, “SHOW INDEX Statement”, and Section 26.3.34, “The INFORMATION_SCHEMA STATISTICS Table”.

For `InnoDB` tables, `ANALYZE TABLE` determines index cardinality by performing random dives on each of the index trees and updating index cardinality estimates accordingly. Because these are only estimates, repeated runs of `ANALYZE TABLE` could produce different numbers. This makes `ANALYZE TABLE` fast on `InnoDB` tables but not 100% accurate because it does not take all rows into account.

You can make the statistics collected by `ANALYZE TABLE` more precise and more stable by enabling `innodb_stats_persistent`, as explained in Section 15.8.10.1, “Configuring Persistent Optimizer Statistics Parameters”. When `innodb_stats_persistent` is enabled, it is important to run `ANALYZE TABLE` after major changes to index column data, as statistics are not recalculated periodically (such as after a server restart).

If `innodb_stats_persistent` is enabled, you can change the number of random dives by modifying the `innodb_stats_persistent_sample_pages` system variable. If `innodb_stats_persistent` is disabled, modify `innodb_stats_transient_sample_pages` instead.

For more information about key distribution analysis in `InnoDB`, see Section 15.8.10.1, “Configuring Persistent Optimizer Statistics Parameters”, and Section 15.8.10.3, “Estimating ANALYZE TABLE Complexity for InnoDB Tables”.

MySQL uses index cardinality estimates in join optimization. If a join is not optimized in the right way, try running `ANALYZE TABLE`. In the few cases that `ANALYZE TABLE` does not produce values good enough for your particular tables, you can use `FORCE INDEX` with your queries to force the use of a particular index, or set the `max_seeks_for_key` system variable to ensure that MySQL prefers index lookups over table scans. See Section B.3.5, “Optimizer-Related Issues”.

Histogram Statistics Analysis

`ANALYZE TABLE` with the `HISTOGRAM` clause enables management of histogram statistics for table column values. For information about histogram statistics, see Section 8.9.6, “Optimizer Statistics”.

These histogram operations are available:

- `ANALYZE TABLE` with an `UPDATE HISTOGRAM` clause generates histogram statistics for the named table columns and stores them in the data dictionary. Only one table name is permitted for this syntax.

  The optional `WITH N BUCKETS` clauses specifies the number of buckets for the histogram. The value of *`N`* must be an integer in the range from 1 to 1024. If this clause is omitted, the number of buckets is 100.

- `ANALYZE TABLE` with a `DROP HISTOGRAM` clause removes histogram statistics for the named table columns from the data dictionary. Only one table name is permitted for this syntax.

Stored histogram management statements affect only the named columns. Consider these statements:

```
ANALYZE TABLE t UPDATE HISTOGRAM ON c1, c2, c3 WITH 10 BUCKETS;
```

```
ANALYZE TABLE t UPDATE HISTOGRAM ON c1, c3 WITH 10 BUCKETS;
ANALYZE TABLE t DROP HISTOGRAM ON c2;
```

The first statement updates the histograms for columns `c1`, `c2`, and `c3`, replacing any existing histograms for those columns. The second statement updates the histograms for `c1` and `c3`, leaving the `c2` histogram unaffected. The third statement removes the histogram for `c2`, leaving those for `c1` and `c3` unaffected.

When sampling user data as part of building a histogram, not all values are read; this may lead to missing some values considered important. In such cases, it might be useful to modify the histogram, or to set your own histogram explicitly based on your own criteria, such as the complete data set. MySQL 8.0.31 adds support for `ANALYZE TABLE tbl_name UPDATE HISTOGRAM ON col_name USING DATA 'json_data'` for updating a column of the histogram table with data supplied in the same JSON format used to display `HISTOGRAM` column values from the Information Schema `COLUMN_STATISTICS` table. Only one column can be modified when updating the histogram with JSON data.

We can illustrate the use of `USING DATA` by first generating a histogram on column `c1` of table `t`, like this:

```
mysql> ANALYZE TABLE t UPDATE HISTOGRAM ON c1;
+--------+-----------+----------+-------------------------------------------+
| Table  | Op        | Msg_type | Msg_text                                  |
+--------+-----------+----------+-------------------------------------------+
| mydb.t | histogram | status   | Histogram statistics created for column 'c1'. |
+--------+-----------+----------+-------------------------------------------+
```

We can see the histogram generated in the `COLUMN_STATISTICS` table:

```
mysql> TABLE information_schema.column_statistics\G
*************************** 1. row ***************************
SCHEMA_NAME: mydb
 TABLE_NAME: t
COLUMN_NAME: c1
  HISTOGRAM: {"buckets": [[206, 0.0625], [456, 0.125], [608, 0.1875]],
"data-type": "int", "null-values": 0.0, "collation-id": 8, "last-updated":
"2022-10-11 16:13:14.563319", "sampling-rate": 1.0, "histogram-type":
"singleton", "number-of-buckets-specified": 100}
```

Now we drop the histogram, and when we check `COLUMN_STATISTICS`, it is now empty:

```
mysql> ANALYZE TABLE t DROP HISTOGRAM ON c1;
+--------+-----------+----------+-------------------------------------------+
| Table  | Op        | Msg_type | Msg_text                                  |
+--------+-----------+----------+-------------------------------------------+
| mydb.t | histogram | status   | Histogram statistics removed for column 'c1'. |
+--------+-----------+----------+-------------------------------------------+

mysql> TABLE information_schema.column_statistics\G
Empty set (0.00 sec)
```

We can restore the dropped histogram by inserting its JSON representation obtained previously from the `HISTOGRAM` column of the `COLUMN_STATISTICS` table, and when we query that table again, we can see that the histogram has been restored to its previous state:

```
mysql> ANALYZE TABLE t UPDATE HISTOGRAM ON c1
    ->     USING DATA '{"buckets": [[206, 0.0625], [456, 0.125], [608, 0.1875]],
    ->                 "data-type": "int", "null-values": 0.0, "collation-id":
    ->                 8, "last-updated": "2022-10-11 16:13:14.563319",
    ->                 "sampling-rate": 1.0, "histogram-type": "singleton",
    ->                 "number-of-buckets-specified": 100}';
+--------+-----------+----------+-------------------------------------------+
| Table  | Op        | Msg_type | Msg_text                                  |
+--------+-----------+----------+-------------------------------------------+
| mydb.t | histogram | status   | Histogram statistics created for column 'c1'. |
+--------+-----------+----------+-------------------------------------------+

mysql> TABLE information_schema.column_statistics\G
```

```
*************************** 1. row ***************************
SCHEMA_NAME: mydb
 TABLE_NAME: t
COLUMN_NAME: c1
  HISTOGRAM: {"buckets": [[206, 0.0625], [456, 0.125], [608, 0.1875]],
"data-type": "int", "null-values": 0.0, "collation-id": 8, "last-updated":
"2022-10-11 16:13:14.563319", "sampling-rate": 1.0, "histogram-type":
"singleton", "number-of-buckets-specified": 100}
```

Histogram generation is not supported for encrypted tables (to avoid exposing data in the statistics) or `TEMPORARY` tables.

Histogram generation applies to columns of all data types except geometry types (spatial data) and `JSON`.

Histograms can be generated for stored and virtual generated columns.

Histograms cannot be generated for columns that are covered by single-column unique indexes.

Histogram management statements attempt to perform as much of the requested operation as possible, and report diagnostic messages for the remainder. For example, if an `UPDATE HISTOGRAM` statement names multiple columns, but some of them do not exist or have an unsupported data type, histograms are generated for the other columns, and messages are produced for the invalid columns.

Histograms are affected by these DDL statements:

- `DROP TABLE` removes histograms for columns in the dropped table.
- `DROP DATABASE` removes histograms for any table in the dropped database because the statement drops all tables in the database.
- `RENAME TABLE` does not remove histograms. Instead, it renames histograms for the renamed table to be associated with the new table name.
- `ALTER TABLE` statements that remove or modify a column remove histograms for that column.
- `ALTER TABLE ... CONVERT TO CHARACTER SET` removes histograms for character columns because they are affected by the change of character set. Histograms for noncharacter columns remain unaffected.

The `histogram_generation_max_mem_size` system variable controls the maximum amount of memory available for histogram generation. The global and session values may be set at runtime.

Changing the global `histogram_generation_max_mem_size` value requires privileges sufficient to set global system variables. Changing the session `histogram_generation_max_mem_size` value requires privileges sufficient to set restricted session system variables. See Section 5.1.9.1, “System Variable Privileges”.

If the estimated amount of data to be read into memory for histogram generation exceeds the limit defined by `histogram_generation_max_mem_size`, MySQL samples the data rather than reading all of it into memory. Sampling is evenly distributed over the entire table. MySQL uses `SYSTEM` sampling, which is a page-level sampling method.

The `sampling-rate` value in the `HISTOGRAM` column of the Information Schema `COLUMN_STATISTICS` table can be queried to determine the fraction of data that was sampled to create the histogram. The `sampling-rate` is a number between 0.0 and 1.0. A value of 1 means that all of the data was read (no sampling).

The following example demonstrates sampling. To ensure that the amount of data exceeds the `histogram_generation_max_mem_size` limit for the purpose of the example, the limit is set to a low value (2000000 bytes) prior to generating histogram statistics for the `birth_date` column of the `employees` table.

```
mysql> SET histogram_generation_max_mem_size = 2000000;

mysql> USE employees;

mysql> ANALYZE TABLE employees UPDATE HISTOGRAM ON birth_date WITH 16 BUCKETS\G
*************************** 1. row ***************************
   Table: employees.employees
      Op: histogram
Msg_type: status
Msg_text: Histogram statistics created for column 'birth_date'.

mysql> SELECT HISTOGRAM->>'$."sampling-rate"'
       FROM INFORMATION_SCHEMA.COLUMN_STATISTICS
       WHERE TABLE_NAME = "employees"
       AND COLUMN_NAME = "birth_date";
+---------------------------------+
| HISTOGRAM->>'$."sampling-rate"' |
+---------------------------------+
| 0.0491431208869665              |
+---------------------------------+
```

A `sampling-rate` value of 0.0491431208869665 means that approximately 4.9% of the data from the `birth_date` column was read into memory for generating histogram statistics.

As of MySQL 8.0.19, the `InnoDB` storage engine provides its own sampling implementation for data stored in `InnoDB` tables. The default sampling implementation used by MySQL when storage engines do not provide their own requires a full table scan, which is costly for large tables. The `InnoDB` sampling implementation improves sampling performance by avoiding full table scans.

The `sampled_pages_read` and `sampled_pages_skipped` `INNODB_METRICS` counters can be used to monitor sampling of `InnoDB` data pages. (For general `INNODB_METRICS` counter usage information, see Section 26.4.21, “The INFORMATION_SCHEMA INNODB_METRICS Table”.)

The following example demonstrates sampling counter usage, which requires enabling the counters prior to generating histogram statistics.

```
mysql> SET GLOBAL innodb_monitor_enable = 'sampled%';

mysql> USE employees;

mysql> ANALYZE TABLE employees UPDATE HISTOGRAM ON birth_date WITH 16 BUCKETS\G
*************************** 1. row ***************************
   Table: employees.employees
      Op: histogram
Msg_type: status
Msg_text: Histogram statistics created for column 'birth_date'.

mysql> USE INFORMATION_SCHEMA;

mysql> SELECT NAME, COUNT FROM INNODB_METRICS WHERE NAME LIKE 'sampled%'\G
*************************** 1. row ***************************
 NAME: sampled_pages_read
COUNT: 43
*************************** 2. row ***************************
 NAME: sampled_pages_skipped
COUNT: 843
```

This formula approximates a sampling rate based on the sampling counter data:

```
sampling rate = sampled_page_read/(sampled_pages_read + sampled_pages_skipped)
```

A sampling rate based on sampling counter data is roughly the same as the `sampling-rate` value in the `HISTOGRAM` column of the Information Schema `COLUMN_STATISTICS` table.

For information about memory allocations performed for histogram generation, monitor the Performance Schema `memory/sql/histograms` instrument. See Section 27.12.20.10, “Memory Summary Tables”.

Other Considerations

`ANALYZE TABLE` clears table statistics from the Information Schema `INNODB_TABLESTATS` table and sets the `STATS_INITIALIZED` column to `Uninitialized`. Statistics are collected again the next time the table is accessed.

13.7.3.2 CHECK TABLE Statement

```
CHECK TABLE tbl_name [, tbl_name] ... [option] ...

option: {
    FOR UPGRADE
  | QUICK
  | FAST
  | MEDIUM
  | EXTENDED
  | CHANGED
}
```

`CHECK TABLE` checks a table or tables for errors. `CHECK TABLE` can also check views for problems, such as tables that are referenced in the view definition that no longer exist.

To check a table, you must have some privilege for it.

`CHECK TABLE` works for `InnoDB`, `MyISAM`, `ARCHIVE`, and `CSV` tables.

Before running `CHECK TABLE` on `InnoDB` tables, see CHECK TABLE Usage Notes for InnoDB Tables.

`CHECK TABLE` is supported for partitioned tables, and you can use `ALTER TABLE ... CHECK PARTITION` to check one or more partitions; for more information, see Section 13.1.9, “ALTER TABLE Statement”, and Section 24.3.4, “Maintenance of Partitions”.

`CHECK TABLE` ignores virtual generated columns that are not indexed.

- CHECK TABLE Output
- Checking Version Compatibility
- Checking Data Consistency
- CHECK TABLE Usage Notes for InnoDB Tables
- CHECK TABLE Usage Notes for MyISAM Tables

CHECK TABLE Output

`CHECK TABLE` returns a result set with the columns shown in the following table.

Column	Value
`Table`	The table name
`Op`	Always `check`
`Msg_type`	`status`, `error`, `info`, `note`, or `warning`
`Msg_text`	An informational message

The statement might produce many rows of information for each checked table. The last row has a `Msg_type` value of `status` and the `Msg_text` normally should be `OK`. `Table is already up to date` means that the storage engine for the table indicated that there was no need to check the table.

Checking Version Compatibility

The `FOR UPGRADE` option checks whether the named tables are compatible with the current version of MySQL. With `FOR UPGRADE`, the server checks each table to determine whether there have been any incompatible changes in any of the table's data types or indexes since the table was created. If not,

the check succeeds. Otherwise, if there is a possible incompatibility, the server runs a full check on the table (which might take some time).

Incompatibilities might occur because the storage format for a data type has changed or because its sort order has changed. Our aim is to avoid these changes, but occasionally they are necessary to correct problems that would be worse than an incompatibility between releases.

`FOR UPGRADE` discovers these incompatibilities:

- The indexing order for end-space in `TEXT` columns for `InnoDB` and `MyISAM` tables changed between MySQL 4.1 and 5.0.
- The storage method of the new `DECIMAL` data type changed between MySQL 5.0.3 and 5.0.5.
- Changes are sometimes made to character sets or collations that require table indexes to be rebuilt. For details about such changes, see Section 2.10.4, “Changes in MySQL 8.0”. For information about rebuilding tables, see Section 2.10.13, “Rebuilding or Repairing Tables or Indexes”.
- MySQL 8.0 does not support the 2-digit `YEAR(2)` data type permitted in older versions of MySQL. For tables containing `YEAR(2)` columns, `CHECK TABLE` recommends `REPAIR TABLE`, which converts 2-digit `YEAR(2)` columns to 4-digit `YEAR` columns.
- Trigger creation time is maintained.
- A table is reported as needing a rebuild if it contains old temporal columns in pre-5.6.4 format (`TIME`, `DATETIME`, and `TIMESTAMP` columns without support for fractional seconds precision) and the `avoid_temporal_upgrade` system variable is disabled. This helps the MySQL upgrade procedure detect and upgrade tables containing old temporal columns. If `avoid_temporal_upgrade` is enabled, `FOR UPGRADE` ignores the old temporal columns present in the table; consequently, the upgrade procedure does not upgrade them.

  To check for tables that contain such temporal columns and need a rebuild, disable `avoid_temporal_upgrade` before executing `CHECK TABLE ... FOR UPGRADE`.
- Warnings are issued for tables that use nonnative partitioning because nonnative partitioning is removed in MySQL 8.0. See Chapter 24, *Partitioning*.

Checking Data Consistency

The following table shows the other check options that can be given. These options are passed to the storage engine, which may use or ignore them.

Type	Meaning
`QUICK`	Do not scan the rows to check for incorrect links. Applies to `InnoDB` and `MyISAM` tables and views.
`FAST`	Check only tables that have not been closed properly. Ignored for `InnoDB`; applies only to `MyISAM` tables and views.
`CHANGED`	Check only tables that have been changed since the last check or that have not been closed properly. Ignored for `InnoDB`; applies only to `MyISAM` tables and views.
`MEDIUM`	Scan rows to verify that deleted links are valid. This also calculates a key checksum for the rows and verifies this with a calculated checksum for the keys. Ignored for `InnoDB`; applies only to `MyISAM` tables and views.
`EXTENDED`	Do a full key lookup for all keys for each row. This ensures that the table is 100% consistent, but

Type	Meaning
	takes a long time. Ignored for `InnoDB`; applies only to `MyISAM` tables and views.

You can combine check options, as in the following example that does a quick check on the table to determine whether it was closed properly:

```
CHECK TABLE test_table FAST QUICK;
```

Note

If `CHECK TABLE` finds no problems with a table that is marked as “corrupted” or “not closed properly”, `CHECK TABLE` may remove the mark.

If a table is corrupted, the problem is most likely in the indexes and not in the data part. All of the preceding check types check the indexes thoroughly and should thus find most errors.

To check a table that you assume is okay, use no check options or the `QUICK` option. The latter should be used when you are in a hurry and can take the very small risk that `QUICK` does not find an error in the data file. (In most cases, under normal usage, MySQL should find any error in the data file. If this happens, the table is marked as “corrupted” and cannot be used until it is repaired.)

`FAST` and `CHANGED` are mostly intended to be used from a script (for example, to be executed from `cron`) to check tables periodically. In most cases, `FAST` is to be preferred over `CHANGED`. (The only case when it is not preferred is when you suspect that you have found a bug in the `MyISAM` code.)

`EXTENDED` is to be used only after you have run a normal check but still get errors from a table when MySQL tries to update a row or find a row by key. This is very unlikely if a normal check has succeeded.

Use of `CHECK TABLE ... EXTENDED` might influence execution plans generated by the query optimizer.

Some problems reported by `CHECK TABLE` cannot be corrected automatically:

- `Found row where the auto_increment column has the value 0`.

  This means that you have a row in the table where the `AUTO_INCREMENT` index column contains the value 0. (It is possible to create a row where the `AUTO_INCREMENT` column is 0 by explicitly setting the column to 0 with an `UPDATE` statement.)

  This is not an error in itself, but could cause trouble if you decide to dump the table and restore it or do an `ALTER TABLE` on the table. In this case, the `AUTO_INCREMENT` column changes value according to the rules of `AUTO_INCREMENT` columns, which could cause problems such as a duplicate-key error.

  To get rid of the warning, execute an `UPDATE` statement to set the column to some value other than 0.

CHECK TABLE Usage Notes for InnoDB Tables

The following notes apply to `InnoDB` tables:

- If `CHECK TABLE` encounters a corrupt page, the server exits to prevent error propagation (Bug #10132). If the corruption occurs in a secondary index but table data is readable, running `CHECK TABLE` can still cause a server exit.
- If `CHECK TABLE` encounters a corrupted `DB_TRX_ID` or `DB_ROLL_PTR` field in a clustered index, `CHECK TABLE` can cause `InnoDB` to access an invalid undo log record, resulting in an MVCC-related server exit.

- If `CHECK TABLE` encounters errors in `InnoDB` tables or indexes, it reports an error, and usually marks the index and sometimes marks the table as corrupted, preventing further use of the index or table. Such errors include an incorrect number of entries in a secondary index or incorrect links.
- If `CHECK TABLE` finds an incorrect number of entries in a secondary index, it reports an error but does not cause a server exit or prevent access to the file.
- `CHECK TABLE` surveys the index page structure, then surveys each key entry. It does not validate the key pointer to a clustered record or follow the path for `BLOB` pointers.
- When an `InnoDB` table is stored in its own `.ibd` file, the first 3 pages of the `.ibd` file contain header information rather than table or index data. The `CHECK TABLE` statement does not detect inconsistencies that affect only the header data. To verify the entire contents of an `InnoDB .ibd` file, use the `innochecksum` command.
- When running `CHECK TABLE` on large `InnoDB` tables, other threads may be blocked during `CHECK TABLE` execution. To avoid timeouts, the semaphore wait threshold (600 seconds) is extended by 2 hours (7200 seconds) for `CHECK TABLE` operations. If `InnoDB` detects semaphore waits of 240 seconds or more, it starts printing `InnoDB` monitor output to the error log. If a lock request extends beyond the semaphore wait threshold, `InnoDB` aborts the process. To avoid the possibility of a semaphore wait timeout entirely, run `CHECK TABLE QUICK` instead of `CHECK TABLE`.
- `CHECK TABLE` functionality for `InnoDB SPATIAL` indexes includes an R-tree validity check and a check to ensure that the R-tree row count matches the clustered index.
- `CHECK TABLE` supports secondary indexes on virtual generated columns, which are supported by `InnoDB`.
- As of MySQL 8.0.14, `InnoDB` supports parallel clustered index reads, which can improve `CHECK TABLE` performance. `InnoDB` reads the clustered index twice during a `CHECK TABLE` operation. The second read can be performed in parallel. The `innodb_parallel_read_threads` session variable must be set to a value greater than 1 for parallel clustered index reads to occur. The default value is 4. The actual number of threads used to perform a parallel clustered index read is determined by the `innodb_parallel_read_threads` setting or the number of index subtrees to scan, whichever is smaller.

CHECK TABLE Usage Notes for MyISAM Tables

The following notes apply to `MyISAM` tables:

- `CHECK TABLE` updates key statistics for `MyISAM` tables.
- If `CHECK TABLE` output does not return `OK` or `Table is already up to date`, you should normally run a repair of the table. See Section 7.6, “MyISAM Table Maintenance and Crash Recovery”.
- If none of the `CHECK TABLE` options `QUICK`, `MEDIUM`, or `EXTENDED` are specified, the default check type for dynamic-format `MyISAM` tables is `MEDIUM`. This has the same result as running `myisamchk --medium-check tbl_name` on the table. The default check type also is `MEDIUM` for static-format `MyISAM` tables, unless `CHANGED` or `FAST` is specified. In that case, the default is `QUICK`. The row scan is skipped for `CHANGED` and `FAST` because the rows are very seldom corrupted.

13.7.3.3 CHECKSUM TABLE Statement

```
CHECKSUM TABLE tbl_name [, tbl_name] ... [QUICK | EXTENDED]
```

`CHECKSUM TABLE` reports a checksum for the contents of a table. You can use this statement to verify that the contents are the same before and after a backup, rollback, or other operation that is intended to put the data back to a known state.

This statement requires the `SELECT` privilege for the table.

This statement is not supported for views. If you run `CHECKSUM TABLE` against a view, the `Checksum` value is always `NULL`, and a warning is returned.

For a nonexistent table, `CHECKSUM TABLE` returns `NULL` and generates a warning.

During the checksum operation, the table is locked with a read lock for `InnoDB` and `MyISAM`.

Performance Considerations

By default, the entire table is read row by row and the checksum is calculated. For large tables, this could take a long time, thus you would only perform this operation occasionally. This row-by-row calculation is what you get with the `EXTENDED` clause, with `InnoDB` and all other storage engines other than `MyISAM`, and with `MyISAM` tables not created with the `CHECKSUM=1` clause.

For `MyISAM` tables created with the `CHECKSUM=1` clause, `CHECKSUM TABLE` or `CHECKSUM TABLE ... QUICK` returns the “live” table checksum that can be returned very fast. If the table does not meet all these conditions, the `QUICK` method returns `NULL`. The `QUICK` method is not supported with `InnoDB` tables. See Section 13.1.20, “CREATE TABLE Statement” for the syntax of the `CHECKSUM` clause.

The checksum value depends on the table row format. If the row format changes, the checksum also changes. For example, the storage format for temporal types such as `TIME`, `DATETIME`, and `TIMESTAMP` changed in MySQL 5.6 prior to MySQL 5.6.5, so if a 5.5 table is upgraded to MySQL 5.6, the checksum value may change.

Important

If the checksums for two tables are different, then it is almost certain that the tables are different in some way. However, because the hashing function used by `CHECKSUM TABLE` is not guaranteed to be collision-free, there is a slight chance that two tables which are not identical can produce the same checksum.

13.7.3.4 OPTIMIZE TABLE Statement

```
OPTIMIZE [NO_WRITE_TO_BINLOG | LOCAL]
    TABLE tbl_name [, tbl_name] ...
```

`OPTIMIZE TABLE` reorganizes the physical storage of table data and associated index data, to reduce storage space and improve I/O efficiency when accessing the table. The exact changes made to each table depend on the storage engine used by that table.

Use `OPTIMIZE TABLE` in these cases, depending on the type of table:

- After doing substantial insert, update, or delete operations on an `InnoDB` table that has its own .ibd file because it was created with the `innodb_file_per_table` option enabled. The table and indexes are reorganized, and disk space can be reclaimed for use by the operating system.
- After doing substantial insert, update, or delete operations on columns that are part of a `FULLTEXT` index in an `InnoDB` table. Set the configuration option `innodb_optimize_fulltext_only=1` first. To keep the index maintenance period to a reasonable time, set the `innodb_ft_num_word_optimize` option to specify how many words to update in the search index, and run a sequence of `OPTIMIZE TABLE` statements until the search index is fully updated.
- After deleting a large part of a `MyISAM` or `ARCHIVE` table, or making many changes to a `MyISAM` or `ARCHIVE` table with variable-length rows (tables that have `VARCHAR`, `VARBINARY`, `BLOB`, or `TEXT` columns). Deleted rows are maintained in a linked list and subsequent `INSERT` operations reuse old row positions. You can use `OPTIMIZE TABLE` to reclaim the unused space and to defragment the data file. After extensive changes to a table, this statement may also improve performance of statements that use the table, sometimes significantly.

This statement requires `SELECT` and `INSERT` privileges for the table.

`OPTIMIZE TABLE` works for `InnoDB`, `MyISAM`, and `ARCHIVE` tables. `OPTIMIZE TABLE` is also supported for dynamic columns of in-memory `NDB` tables. It does not work for fixed-width columns of in-memory tables, nor does it work for Disk Data tables. The performance of `OPTIMIZE` on NDB Cluster tables can be tuned using `--ndb-optimization-delay`, which controls the length of time to wait between processing batches of rows by `OPTIMIZE TABLE`. For more information, see Section 23.2.7.11, “Previous NDB Cluster Issues Resolved in NDB Cluster 8.0”.

For NDB Cluster tables, `OPTIMIZE TABLE` can be interrupted by (for example) killing the SQL thread performing the `OPTIMIZE` operation.

By default, `OPTIMIZE TABLE` does *not* work for tables created using any other storage engine and returns a result indicating this lack of support. You can make `OPTIMIZE TABLE` work for other storage engines by starting `mysqld` with the `--skip-new` option. In this case, `OPTIMIZE TABLE` is just mapped to `ALTER TABLE`.

This statement does not work with views.

`OPTIMIZE TABLE` is supported for partitioned tables. For information about using this statement with partitioned tables and table partitions, see Section 24.3.4, “Maintenance of Partitions”.

By default, the server writes `OPTIMIZE TABLE` statements to the binary log so that they replicate to replicas. To suppress logging, specify the optional `NO_WRITE_TO_BINLOG` keyword or its alias `LOCAL`.

- OPTIMIZE TABLE Output
- InnoDB Details
- MyISAM Details
- Other Considerations

OPTIMIZE TABLE Output

`OPTIMIZE TABLE` returns a result set with the columns shown in the following table.

Column	Value
`Table`	The table name
`Op`	Always `optimize`
`Msg_type`	`status`, `error`, `info`, `note`, or `warning`
`Msg_text`	An informational message

`OPTIMIZE TABLE` table catches and throws any errors that occur while copying table statistics from the old file to the newly created file. For example. if the user ID of the owner of the `.MYD` or `.MYI` file is different from the user ID of the `mysqld` process, `OPTIMIZE TABLE` generates a "cannot change ownership of the file" error unless `mysqld` is started by the `root` user.

InnoDB Details

For `InnoDB` tables, `OPTIMIZE TABLE` is mapped to `ALTER TABLE ... FORCE`, which rebuilds the table to update index statistics and free unused space in the clustered index. This is displayed in the output of `OPTIMIZE TABLE` when you run it on an `InnoDB` table, as shown here:

```
mysql> OPTIMIZE TABLE foo;
+----------+----------+----------+-------------------------------------------------------------------+
| Table    | Op       | Msg_type | Msg_text                                                          |
+----------+----------+----------+-------------------------------------------------------------------+
| test.foo | optimize | note     | Table does not support optimize, doing recreate + analyze instead |
| test.foo | optimize | status   | OK                                                                |
+----------+----------+----------+-------------------------------------------------------------------+
```

`OPTIMIZE TABLE` uses online DDL for regular and partitioned `InnoDB` tables, which reduces downtime for concurrent DML operations. The table rebuild triggered by `OPTIMIZE TABLE` is

completed in place. An exclusive table lock is only taken briefly during the prepare phase and the commit phase of the operation. During the prepare phase, metadata is updated and an intermediate table is created. During the commit phase, table metadata changes are committed.

`OPTIMIZE TABLE` rebuilds the table using the table copy method under the following conditions:

- When the `old_alter_table` system variable is enabled.
- When the server is started with the `--skip-new` option.

`OPTIMIZE TABLE` using online DDL is not supported for `InnoDB` tables that contain `FULLTEXT` indexes. The table copy method is used instead.

`InnoDB` stores data using a page-allocation method and does not suffer from fragmentation in the same way that legacy storage engines (such as `MyISAM`) do. When considering whether or not to run optimize, consider the workload of transactions that your server is expected to process:

- Some level of fragmentation is expected. `InnoDB` only fills pages 93% full, to leave room for updates without having to split pages.
- Delete operations might leave gaps that leave pages less filled than desired, which could make it worthwhile to optimize the table.
- Updates to rows usually rewrite the data within the same page, depending on the data type and row format, when sufficient space is available. See Section 15.9.1.5, “How Compression Works for InnoDB Tables” and Section 15.10, “InnoDB Row Formats”.
- High-concurrency workloads might leave gaps in indexes over time, as `InnoDB` retains multiple versions of the same data due through its MVCC mechanism. See Section 15.3, “InnoDB Multi-Versioning”.

MyISAM Details

For `MyISAM` tables, `OPTIMIZE TABLE` works as follows:

1. If the table has deleted or split rows, repair the table.
2. If the index pages are not sorted, sort them.
3. If the table's statistics are not up to date (and the repair could not be accomplished by sorting the index), update them.

Other Considerations

`OPTIMIZE TABLE` is performed online for regular and partitioned `InnoDB` tables. Otherwise, MySQL locks the table during the time `OPTIMIZE TABLE` is running.

`OPTIMIZE TABLE` does not sort R-tree indexes, such as spatial indexes on `POINT` columns. (Bug #23578)

13.7.3.5 REPAIR TABLE Statement

```
REPAIR [NO_WRITE_TO_BINLOG | LOCAL]
    TABLE tbl_name [, tbl_name] ...
    [QUICK] [EXTENDED] [USE_FRM]
```

`REPAIR TABLE` repairs a possibly corrupted table, for certain storage engines only.

This statement requires `SELECT` and `INSERT` privileges for the table.

Although normally you should never have to run `REPAIR TABLE`, if disaster strikes, this statement is very likely to get back all your data from a `MyISAM` table. If your tables become corrupted often, try to find the reason for it, to eliminate the need to use `REPAIR TABLE`. See Section B.3.3.3, “What to Do If MySQL Keeps Crashing”, and Section 16.2.4, “MyISAM Table Problems”.

`REPAIR TABLE` checks the table to see whether an upgrade is required. If so, it performs the upgrade, following the same rules as `CHECK TABLE ... FOR UPGRADE`. See Section 13.7.3.2, "CHECK TABLE Statement", for more information.

Important

- Make a backup of a table before performing a table repair operation; under some circumstances the operation might cause data loss. Possible causes include but are not limited to file system errors. See Chapter 7, *Backup and Recovery*.
- If the server exits during a `REPAIR TABLE` operation, it is essential after restarting it that you immediately execute another `REPAIR TABLE` statement for the table before performing any other operations on it. In the worst case, you might have a new clean index file without information about the data file, and then the next operation you perform could overwrite the data file. This is an unlikely but possible scenario that underscores the value of making a backup first.
- In the event that a table on the source becomes corrupted and you run `REPAIR TABLE` on it, any resulting changes to the original table are *not* propagated to replicas.

- REPAIR TABLE Storage Engine and Partitioning Support
- REPAIR TABLE Options
- REPAIR TABLE Output
- Table Repair Considerations

REPAIR TABLE Storage Engine and Partitioning Support

`REPAIR TABLE` works for `MyISAM`, `ARCHIVE`, and `CSV` tables. For `MyISAM` tables, it has the same effect as `myisamchk --recover tbl_name` by default. This statement does not work with views.

`REPAIR TABLE` is supported for partitioned tables. However, the `USE_FRM` option cannot be used with this statement on a partitioned table.

You can use `ALTER TABLE ... REPAIR PARTITION` to repair one or more partitions; for more information, see Section 13.1.9, "ALTER TABLE Statement", and Section 24.3.4, "Maintenance of Partitions".

REPAIR TABLE Options

- `NO_WRITE_TO_BINLOG` or `LOCAL`

  By default, the server writes `REPAIR TABLE` statements to the binary log so that they replicate to replicas. To suppress logging, specify the optional `NO_WRITE_TO_BINLOG` keyword or its alias `LOCAL`.

- `QUICK`

  If you use the `QUICK` option, `REPAIR TABLE` tries to repair only the index file, and not the data file. This type of repair is like that done by `myisamchk --recover --quick`.

- `EXTENDED`

  If you use the `EXTENDED` option, MySQL creates the index row by row instead of creating one index at a time with sorting. This type of repair is like that done by `myisamchk --safe-recover`.

- `USE_FRM`

  The `USE_FRM` option is available for use if the `.MYI` index file is missing or if its header is corrupted. This option tells MySQL not to trust the information in the `.MYI` file header and to re-create it using information from the data dictionary. This kind of repair cannot be done with `myisamchk`.

Caution

Use the `USE_FRM` option *only* if you cannot use regular `REPAIR` modes. Telling the server to ignore the `.MYI` file makes important table metadata stored in the `.MYI` unavailable to the repair process, which can have deleterious consequences:

- The current `AUTO_INCREMENT` value is lost.
- The link to deleted records in the table is lost, which means that free space for deleted records remains unoccupied thereafter.
- The `.MYI` header indicates whether the table is compressed. If the server ignores this information, it cannot tell that a table is compressed and repair can cause change or loss of table contents. This means that `USE_FRM` should not be used with compressed tables. That should not be necessary, anyway: Compressed tables are read only, so they should not become corrupt.

If you use `USE_FRM` for a table that was created by a different version of the MySQL server than the one you are currently running, `REPAIR TABLE` does not attempt to repair the table. In this case, the result set returned by `REPAIR TABLE` contains a line with a `Msg_type` value of `error` and a `Msg_text` value of `Failed repairing incompatible .FRM file`.

If `USE_FRM` is used, `REPAIR TABLE` does not check the table to see whether an upgrade is required.

REPAIR TABLE Output

`REPAIR TABLE` returns a result set with the columns shown in the following table.

Column	Value
`Table`	The table name
`Op`	Always `repair`
`Msg_type`	`status`, `error`, `info`, `note`, or `warning`
`Msg_text`	An informational message

The `REPAIR TABLE` statement might produce many rows of information for each repaired table. The last row has a `Msg_type` value of `status` and `Msg_test` normally should be `OK`. For a `MyISAM` table, if you do not get `OK`, you should try repairing it with `myisamchk --safe-recover`. (`REPAIR TABLE` does not implement all the options of `myisamchk`. With `myisamchk --safe-recover`, you can also use options that `REPAIR TABLE` does not support, such as `--max-record-length`.)

`REPAIR TABLE` table catches and throws any errors that occur while copying table statistics from the old corrupted file to the newly created file. For example. if the user ID of the owner of the `.MYD` or `.MYI` file is different from the user ID of the `mysqld` process, `REPAIR TABLE` generates a "cannot change ownership of the file" error unless `mysqld` is started by the `root` user.

Table Repair Considerations

`REPAIR TABLE` upgrades a table if it contains old temporal columns in pre-5.6.4 format (`TIME`, `DATETIME`, and `TIMESTAMP` columns without support for fractional seconds precision) and the

`avoid_temporal_upgrade` system variable is disabled. If `avoid_temporal_upgrade` is enabled, `REPAIR TABLE` ignores the old temporal columns present in the table and does not upgrade them.

To upgrade tables that contain such temporal columns, disable `avoid_temporal_upgrade` before executing `REPAIR TABLE`.

You may be able to increase `REPAIR TABLE` performance by setting certain system variables. See Section 8.6.3, "Optimizing REPAIR TABLE Statements".

13.7.4 Component, Plugin, and Loadable Function Statements

13.7.4.1 CREATE FUNCTION Statement for Loadable Functions

```
CREATE [AGGREGATE] FUNCTION [IF NOT EXISTS] function_name
    RETURNS {STRING|INTEGER|REAL|DECIMAL}
    SONAME shared_library_name
```

This statement loads the loadable function named *`function_name`*. (`CREATE FUNCTION` is also used to created stored functions; see Section 13.1.17, "CREATE PROCEDURE and CREATE FUNCTION Statements".)

A loadable function is a way to extend MySQL with a new function that works like a native (built-in) MySQL function such as `ABS()` or `CONCAT()`. See Adding a Loadable Function.

`function_name` is the name that should be used in SQL statements to invoke the function. The `RETURNS` clause indicates the type of the function's return value. `DECIMAL` is a legal value after `RETURNS`, but currently `DECIMAL` functions return string values and should be written like `STRING` functions.

`IF NOT EXISTS` prevents an error from occurring if there already exists a loadable function with the same name. It does *not* prevent an error from occurring if there already exists a built-in function having the same name. `IF NOT EXISTS` is supported for `CREATE FUNCTION` statements beginning with MySQL 8.0.29. See also Function Name Resolution.

The `AGGREGATE` keyword, if given, signifies that the function is an aggregate (group) function. An aggregate function works exactly like a native MySQL aggregate function such as `SUM()` or `COUNT()`.

`shared_library_name` is the base name of the shared library file containing the code that implements the function. The file must be located in the plugin directory. This directory is given by the value of the `plugin_dir` system variable. For more information, see Section 5.7.1, "Installing and Uninstalling Loadable Functions".

`CREATE FUNCTION` requires the `INSERT` privilege for the `mysql` system schema because it adds a row to the `mysql.func` system table to register the function.

`CREATE FUNCTION` also adds the function to the Performance Schema `user_defined_functions` table that provides runtime information about installed loadable functions. See Section 27.12.21.9, "The user_defined_functions Table".

Note

Like the `mysql.func` system table, the Performance Schema `user_defined_functions` table lists loadable functions installed using `CREATE FUNCTION`. Unlike the `mysql.func` table, the `user_defined_functions` table also lists loadable functions installed automatically by server components or plugins. This difference makes `user_defined_functions` preferable to `mysql.func` for checking which loadable functions are installed.

During the normal startup sequence, the server loads functions registered in the `mysql.func` table. If the server is started with the `--skip-grant-tables` option, functions registered in the table are not loaded and are unavailable.

Note

To upgrade the shared library associated with a loadable function, issue a `DROP FUNCTION` statement, upgrade the shared library, and then issue a `CREATE FUNCTION` statement. If you upgrade the shared library first and then use `DROP FUNCTION`, the server may unexpectedly shut down.

13.7.4.2 DROP FUNCTION Statement for Loadable Functions

```
DROP FUNCTION [IF EXISTS] function_name
```

This statement drops the loadable function named *`function_name`*. (`DROP FUNCTION` is also used to drop stored functions; see Section 13.1.29, "DROP PROCEDURE and DROP FUNCTION Statements".)

`DROP FUNCTION` is the complement of `CREATE FUNCTION`. It requires the `DELETE` privilege for the `mysql` system schema because it removes the row from the `mysql.func` system table that registers the function.

`DROP FUNCTION` also removes the function from the Performance Schema `user_defined_functions` table that provides runtime information about installed loadable functions. See Section 27.12.21.9, "The user_defined_functions Table".

During the normal startup sequence, the server loads functions registered in the `mysql.func` table. Because `DROP FUNCTION` removes the `mysql.func` row for the dropped function, the server does not load the function during subsequent restarts.

`DROP FUNCTION` cannot be used to drop a loadable function that is installed automatically by components or plugins rather than by using `CREATE FUNCTION`. Such a function is also dropped automatically, when the component or plugin that installed it is uninstalled.

Note

To upgrade the shared library associated with a loadable function, issue a `DROP FUNCTION` statement, upgrade the shared library, and then issue a `CREATE FUNCTION` statement. If you upgrade the shared library first and then use `DROP FUNCTION`, the server may unexpectedly shut down.

13.7.4.3 INSTALL COMPONENT Statement

```
INSTALL COMPONENT component_name [, component_name ] ...
```

This statement installs one or more components, which become active immediately. A component provides services that are available to the server and other components; see Section 5.5, "MySQL Components". `INSTALL COMPONENT` requires the `INSERT` privilege for the `mysql.component` system table because it adds a row to that table to register the component.

Example:

```
INSTALL COMPONENT 'file://component1', 'file://component2';
```

A component is named using a URN that begins with `file://` and indicates the base name of the library file that implements the component, located in the directory named by the `plugin_dir` system variable. Component names do not include any platform-dependent file name suffix such as `.so` or `.dll`. (These naming details are subject to change because component name interpretation is itself performed by a service and the component infrastructure makes it possible to replace the default service implementation with alternative implementations.)

If any error occurs, the statement fails and has no effect. For example, this happens if a component name is erroneous, a named component does not exist or is already installed, or component initialization fails.

A loader service handles component loading, which includes adding installed components to the `mysql.component` system table that serves as a registry. For subsequent server restarts, any components listed in `mysql.component` are loaded by the loader service during the startup sequence. This occurs even if the server is started with the `--skip-grant-tables` option.

If a component depends on services not present in the registry and you attempt to install the component without also installing the component or components that provide the services on which it depends, an error occurs:

```
ERROR 3527 (HY000): Cannot satisfy dependency for service 'component_a'
required by component 'component_b'.
```

To avoid this problem, either install all components in the same statement, or install the dependent component after installing any components on which it depends.

Note

For keyring components, do not use `INSTALL COMPONENT`. Instead, configure keyring component loading using a manifest file. See Section 6.4.4.2, “Keyring Component Installation”.

13.7.4.4 INSTALL PLUGIN Statement

```
INSTALL PLUGIN plugin_name SONAME 'shared_library_name'
```

This statement installs a server plugin. It requires the `INSERT` privilege for the `mysql.plugin` system table because it adds a row to that table to register the plugin.

`plugin_name` is the name of the plugin as defined in the plugin descriptor structure contained in the library file (see Plugin Data Structures). Plugin names are not case-sensitive. For maximal compatibility, plugin names should be limited to ASCII letters, digits, and underscore because they are used in C source files, shell command lines, M4 and Bourne shell scripts, and SQL environments.

`shared_library_name` is the name of the shared library that contains the plugin code. The name includes the file name extension (for example, `libmyplugin.so`, `libmyplugin.dll`, or `libmyplugin.dylib`).

The shared library must be located in the plugin directory (the directory named by the `plugin_dir` system variable). The library must be in the plugin directory itself, not in a subdirectory. By default, `plugin_dir` is the `plugin` directory under the directory named by the `pkglibdir` configuration variable, but it can be changed by setting the value of `plugin_dir` at server startup. For example, set its value in a `my.cnf` file:

```
[mysqld]
plugin_dir=/path/to/plugin/directory
```

If the value of `plugin_dir` is a relative path name, it is taken to be relative to the MySQL base directory (the value of the `basedir` system variable).

`INSTALL PLUGIN` loads and initializes the plugin code to make the plugin available for use. A plugin is initialized by executing its initialization function, which handles any setup that the plugin must perform before it can be used. When the server shuts down, it executes the deinitialization function for each plugin that is loaded so that the plugin has a chance to perform any final cleanup.

`INSTALL PLUGIN` also registers the plugin by adding a line that indicates the plugin name and library file name to the `mysql.plugin` system table. During the normal startup sequence, the server loads and initializes plugins registered in `mysql.plugin`. This means that a plugin is installed with `INSTALL PLUGIN` only once, not every time the server starts. If the server is started with the `--skip-grant-tables` option, plugins registered in the `mysql.plugin` table are not loaded and are unavailable.

A plugin library can contain multiple plugins. For each of them to be installed, use a separate `INSTALL PLUGIN` statement. Each statement names a different plugin, but all of them specify the same library name.

`INSTALL PLUGIN` causes the server to read option (`my.cnf`) files just as during server startup. This enables the plugin to pick up any relevant options from those files. It is possible to add plugin options to an option file even before loading a plugin (if the `loose` prefix is used). It is also possible to uninstall a plugin, edit `my.cnf`, and install the plugin again. Restarting the plugin this way enables it to the new option values without a server restart.

For options that control individual plugin loading at server startup, see Section 5.6.1, “Installing and Uninstalling Plugins”. If you need to load plugins for a single server startup when the `--skip-grant-tables` option is given (which tells the server not to read system tables), use the `--plugin-load` option. See Section 5.1.7, “Server Command Options”.

To remove a plugin, use the `UNINSTALL PLUGIN` statement.

For additional information about plugin loading, see Section 5.6.1, “Installing and Uninstalling Plugins”.

To see what plugins are installed, use the `SHOW PLUGINS` statement or query the `INFORMATION_SCHEMA` the `PLUGINS` table.

If you recompile a plugin library and need to reinstall it, you can use either of the following methods:

- Use `UNINSTALL PLUGIN` to uninstall all plugins in the library, install the new plugin library file in the plugin directory, and then use `INSTALL PLUGIN` to install all plugins in the library. This procedure has the advantage that it can be used without stopping the server. However, if the plugin library contains many plugins, you must issue many `INSTALL PLUGIN` and `UNINSTALL PLUGIN` statements.
- Stop the server, install the new plugin library file in the plugin directory, and restart the server.

13.7.4.5 UNINSTALL COMPONENT Statement

```
UNINSTALL COMPONENT component_name [, component_name ] ...
```

This statement deactivates and uninstalls one or more components. A component provides services that are available to the server and other components; see Section 5.5, “MySQL Components”. `UNINSTALL COMPONENT` is the complement of `INSTALL COMPONENT`. It requires the `DELETE` privilege for the `mysql.component` system table because it removes the row from that table that registers the component.

Example:

```
UNINSTALL COMPONENT 'file://component1', 'file://component2';
```

For information about component naming, see Section 13.7.4.3, “INSTALL COMPONENT Statement”.

If any error occurs, the statement fails and has no effect. For example, this happens if a component name is erroneous, a named component is not installed, or cannot be uninstalled because other installed components depend on it.

A loader service handles component unloading, which includes removing uninstalled components from the `mysql.component` system table that serves as a registry. As a result, unloaded components are not loaded during the startup sequence for subsequent server restarts.

Note

This statement has no effect for keyring components, which are loaded using a manifest file and cannot be uninstalled. See Section 6.4.4.2, “Keyring Component Installation”.

13.7.4.6 UNINSTALL PLUGIN Statement

```
UNINSTALL PLUGIN plugin_name
```

This statement removes an installed server plugin. `UNINSTALL PLUGIN` is the complement of `INSTALL PLUGIN`. It requires the `DELETE` privilege for the `mysql.plugin` system table because it removes the row from that table that registers the plugin.

`plugin_name` must be the name of some plugin that is listed in the `mysql.plugin` table. The server executes the plugin's deinitialization function and removes the row for the plugin from the `mysql.plugin` system table, so that subsequent server restarts do not load and initialize the plugin. `UNINSTALL PLUGIN` does not remove the plugin's shared library file.

You cannot uninstall a plugin if any table that uses it is open.

Plugin removal has implications for the use of associated tables. For example, if a full-text parser plugin is associated with a `FULLTEXT` index on the table, uninstalling the plugin makes the table unusable. Any attempt to access the table results in an error. The table cannot even be opened, so you cannot drop an index for which the plugin is used. This means that uninstalling a plugin is something to do with care unless you do not care about the table contents. If you are uninstalling a plugin with no intention of reinstalling it later and you care about the table contents, you should dump the table with `mysqldump` and remove the `WITH PARSER` clause from the dumped `CREATE TABLE` statement so that you can reload the table later. If you do not care about the table, `DROP TABLE` can be used even if any plugins associated with the table are missing.

For additional information about plugin loading, see Section 5.6.1, “Installing and Uninstalling Plugins”.

13.7.5 CLONE Statement

```
CLONE clone_action

clone_action: {
    LOCAL DATA DIRECTORY [=] 'clone_dir';
  | INSTANCE FROM 'user'@'host':port
    IDENTIFIED BY 'password'
    [DATA DIRECTORY [=] 'clone_dir']
    [REQUIRE [NO] SSL]
}
```

The `CLONE` statement is used to clone data locally or from a remote MySQL server instance. To use `CLONE` syntax, the clone plugin must be installed. See Section 5.6.7, “The Clone Plugin”.

`CLONE LOCAL DATA DIRECTORY` syntax clones data from the local MySQL data directory to a directory on the same server or node where the MySQL server instance runs. The `'clone_dir'` directory is the full path of the local directory that data is cloned to. An absolute path is required. The specified directory must not exist, but the specified path must be an existent path. The MySQL server requires the necessary write access to create the specified directory. For more information, see Section 5.6.7.2, “Cloning Data Locally”.

`CLONE INSTANCE` syntax clones data from a remote MySQL server instance (the donor) and transfers it to the MySQL instance where the cloning operation was initiated (the recipient).

- *`user`* is the clone user on the donor MySQL server instance.
- *`host`* is the `hostname` address of the donor MySQL server instance. Internet Protocol version 6 (IPv6) address format is not supported. An alias to the IPv6 address can be used instead. An IPv4 address can be used as is.
- *`port`* is the `port` number of the donor MySQL server instance. (The X Protocol port specified by `mysqlx_port` is not supported. Connecting to the donor MySQL server instance through MySQL Router is also not supported.)
- `IDENTIFIED BY '`*`password`*`'` specifies the password of the clone user on the donor MySQL server instance.
- `DATA DIRECTORY [=] '`*`clone_dir`*`'` is an optional clause used to specify a directory on the recipient for the data you are cloning. Use this option if you do not want to remove existing data

in the recipient data directory. An absolute path is required, and the directory must not exist. The MySQL server must have the necessary write access to create the directory.

When the optional `DATA DIRECTORY [=] 'clone_dir'` clause is not used, a cloning operation removes existing data in the recipient data directory, replaces it with the cloned data, and automatically restarts the server afterward.

- `[REQUIRE [NO] SSL]` explicitly specifies whether an encrypted connection is to be used or not when transferring cloned data over the network. An error is returned if the explicit specification cannot be satisfied. If an SSL clause is not specified, clone attempts to establish an encrypted connection by default, falling back to an insecure connection if the secure connection attempt fails. A secure connection is required when cloning encrypted data regardless of whether this clause is specified. For more information, see Configuring an Encrypted Connection for Cloning.

For additional information about cloning data from a remote MySQL server instance, see Section 5.6.7.3, “Cloning Remote Data”.

13.7.6 SET Statements

The `SET` statement has several forms. Descriptions for those forms that are not associated with a specific server capability appear in subsections of this section:

- `SET var_name = value` enables you to assign values to variables that affect the operation of the server or clients. See Section 13.7.6.1, “SET Syntax for Variable Assignment”.
- `SET CHARACTER SET` and `SET NAMES` assign values to character set and collation variables associated with the current connection to the server. See Section 13.7.6.2, “SET CHARACTER SET Statement”, and Section 13.7.6.3, “SET NAMES Statement”.

Descriptions for the other forms appear elsewhere, grouped with other statements related to the capability they help implement:

- `SET DEFAULT ROLE` and `SET ROLE` set the default role and current role for user accounts. See Section 13.7.1.9, “SET DEFAULT ROLE Statement”, and Section 13.7.1.11, “SET ROLE Statement”.
- `SET PASSWORD` assigns account passwords. See Section 13.7.1.10, “SET PASSWORD Statement”.
- `SET RESOURCE GROUP` assigns threads to a resource group. See Section 13.7.2.4, “SET RESOURCE GROUP Statement”.
- `SET TRANSACTION ISOLATION LEVEL` sets the isolation level for transaction processing. See Section 13.3.7, “SET TRANSACTION Statement”.

13.7.6.1 SET Syntax for Variable Assignment

```
SET variable = expr [, variable = expr] ...

variable: {
    user_var_name
  | param_name
  | local_var_name
  | {GLOBAL | @@GLOBAL.} system_var_name
  | {PERSIST | @@PERSIST.} system_var_name
  | {PERSIST_ONLY | @@PERSIST_ONLY.} system_var_name
  | [SESSION | @@SESSION. | @@] system_var_name
}
```

`SET` syntax for variable assignment enables you to assign values to different types of variables that affect the operation of the server or clients:

- User-defined variables. See Section 9.4, “User-Defined Variables”.
- Stored procedure and function parameters, and stored program local variables. See Section 13.6.4, “Variables in Stored Programs”.

- System variables. See Section 5.1.8, “Server System Variables”. System variables also can be set at server startup, as described in Section 5.1.9, “Using System Variables”.

A `SET` statement that assigns variable values is not written to the binary log, so in replication scenarios it affects only the host on which you execute it. To affect all replication hosts, execute the statement on each host.

The following sections describe `SET` syntax for setting variables. They use the `=` assignment operator, but the `:=` assignment operator is also permitted for this purpose.

- User-Defined Variable Assignment
- Parameter and Local Variable Assignment
- System Variable Assignment
- SET Error Handling
- Multiple Variable Assignment
- System Variable References in Expressions

User-Defined Variable Assignment

User-defined variables are created locally within a session and exist only within the context of that session; see Section 9.4, “User-Defined Variables”.

A user-defined variable is written as *`@var_name`* and is assigned an expression value as follows:

```
SET @var_name = expr;
```

Examples:

```
SET @name = 43;
SET @total_tax = (SELECT SUM(tax) FROM taxable_transactions);
```

As demonstrated by those statements, *`expr`* can range from simple (a literal value) to more complex (the value returned by a scalar subquery).

The Performance Schema `user_variables_by_thread` table contains information about user-defined variables. See Section 27.12.10, “Performance Schema User-Defined Variable Tables”.

Parameter and Local Variable Assignment

`SET` applies to parameters and local variables in the context of the stored object within which they are defined. The following procedure uses the `increment` procedure parameter and `counter` local variable:

```
CREATE PROCEDURE p(increment INT)
BEGIN
  DECLARE counter INT DEFAULT 0;
  WHILE counter < 10 DO
    -- ... do work ...
    SET counter = counter + increment;
  END WHILE;
END;
```

System Variable Assignment

The MySQL server maintains system variables that configure its operation. A system variable can have a global value that affects server operation as a whole, a session value that affects the current session, or both. Many system variables are dynamic and can be changed at runtime using the `SET` statement to affect operation of the current server instance. `SET` can also be used to persist certain system variables to the `mysqld-auto.cnf` file in the data directory, to affect server operation for subsequent startups.

If a SET statement is issued for a sensitive system variable, the query is rewritten to replace the value with "<redacted>" before it is logged to the general log and audit log. This takes place even if secure storage through a keyring component is not available on the server instance.

If you change a session system variable, the value remains in effect within your session until you change the variable to a different value or the session ends. The change has no effect on other sessions.

If you change a global system variable, the value is remembered and used to initialize the session value for new sessions until you change the variable to a different value or the server exits. The change is visible to any client that accesses the global value. However, the change affects the corresponding session value only for clients that connect after the change. The global variable change does not affect the session value for any current client sessions (not even the session within which the global value change occurs).

To make a global system variable setting permanent so that it applies across server restarts, you can persist it to the mysqld-auto.cnf file in the data directory. It is also possible to make persistent configuration changes by manually modifying a my.cnf option file, but that is more cumbersome, and an error in a manually entered setting might not be discovered until much later. SET statements that persist system variables are more convenient and avoid the possibility of malformed settings because settings with syntax errors do not succeed and do not change server configuration. For more information about persisting system variables and the mysqld-auto.cnf file, see Section 5.1.9.3, "Persisted System Variables".

Note

Setting or persisting a global system variable value always requires special privileges. Setting a session system variable value normally requires no special privileges and can be done by any user, although there are exceptions. For more information, see Section 5.1.9.1, "System Variable Privileges".

The following discussion describes the syntax options for setting and persisting system variables:

- To assign a value to a global system variable, precede the variable name by the GLOBAL keyword or the @@GLOBAL. qualifier:

```
SET GLOBAL max_connections = 1000;
SET @@GLOBAL.max_connections = 1000;
```

- To assign a value to a session system variable, precede the variable name by the SESSION or LOCAL keyword, by the @@SESSION., @@LOCAL., or @@ qualifier, or by no keyword or no modifier at all:

```
SET SESSION sql_mode = 'TRADITIONAL';
SET LOCAL sql_mode = 'TRADITIONAL';
SET @@SESSION.sql_mode = 'TRADITIONAL';
SET @@LOCAL.sql_mode = 'TRADITIONAL';
SET @@sql_mode = 'TRADITIONAL';
SET sql_mode = 'TRADITIONAL';
```

  A client can change its own session variables, but not those of any other client.

- To persist a global system variable to the mysqld-auto.cnf option file in the data directory, precede the variable name by the PERSIST keyword or the @@PERSIST. qualifier:

```
SET PERSIST max_connections = 1000;
SET @@PERSIST.max_connections = 1000;
```

  This SET syntax enables you to make configuration changes at runtime that also persist across server restarts. Like SET GLOBAL, SET PERSIST sets the global variable runtime value, but also writes the variable setting to the mysqld-auto.cnf file (replacing any existing variable setting if there is one).

- To persist a global system variable to the `mysqld-auto.cnf` file without setting the global variable runtime value, precede the variable name by the `PERSIST_ONLY` keyword or the `@@PERSIST_ONLY.` qualifier:

  ```
  SET PERSIST_ONLY back_log = 100;
  SET @@PERSIST_ONLY.back_log = 100;
  ```

  Like `PERSIST`, `PERSIST_ONLY` writes the variable setting to `mysqld-auto.cnf`. However, unlike `PERSIST`, `PERSIST_ONLY` does not modify the global variable runtime value. This makes `PERSIST_ONLY` suitable for configuring read-only system variables that can be set only at server startup.

To set a global system variable value to the compiled-in MySQL default value or a session system variable to the current corresponding global value, set the variable to the value `DEFAULT`. For example, the following two statements are identical in setting the session value of `max_join_size` to the current global value:

```
SET @@SESSION.max_join_size = DEFAULT;
SET @@SESSION.max_join_size = @@GLOBAL.max_join_size;
```

Using `SET` to persist a global system variable to a value of `DEFAULT` or to its literal default value assigns the variable its default value and adds a setting for the variable to `mysqld-auto.cnf`. To remove the variable from the file, use `RESET PERSIST`.

Some system variables cannot be persisted or are persist-restricted. See Section 5.1.9.4, “Nonpersistible and Persist-Restricted System Variables”.

A system variable implemented by a plugin can be persisted if the plugin is installed when the `SET` statement is executed. Assignment of the persisted plugin variable takes effect for subsequent server restarts if the plugin is still installed. If the plugin is no longer installed, the plugin variable no longer exists when the server reads the `mysqld-auto.cnf` file. In this case, the server writes a warning to the error log and continues:

```
currently unknown variable 'var_name'
was read from the persisted config file
```

To display system variable names and values:

- Use the `SHOW VARIABLES` statement; see Section 13.7.7.41, “SHOW VARIABLES Statement”.
- Several Performance Schema tables provide system variable information. See Section 27.12.14, “Performance Schema System Variable Tables”.
- The Performance Schema `variables_info` table contains information showing when and by which user each system variable was most recently set. See Section 27.12.14.2, “Performance Schema variables_info Table”.
- The Performance Schema `persisted_variables` table provides an SQL interface to the `mysqld-auto.cnf` file, enabling its contents to be inspected at runtime using `SELECT` statements. See Section 27.12.14.1, “Performance Schema persisted_variables Table”.

SET Error Handling

If any variable assignment in a `SET` statement fails, the entire statement fails and no variables are changed, nor is the `mysqld-auto.cnf` file changed.

`SET` produces an error under the circumstances described here. Most of the examples show `SET` statements that use keyword syntax (for example, `GLOBAL` or `SESSION`), but the principles are also true for statements that use the corresponding modifiers (for example, `@@GLOBAL.` or `@@SESSION.`).

- Use of `SET` (any variant) to set a read-only variable:

  ```
  mysql> SET GLOBAL version = 'abc';
  ```

```
ERROR 1238 (HY000): Variable 'version' is a read only variable
```

- Use of `GLOBAL`, `PERSIST`, or `PERSIST_ONLY` to set a variable that has only a session value:

```
mysql> SET GLOBAL sql_log_bin = ON;
ERROR 1228 (HY000): Variable 'sql_log_bin' is a SESSION
variable and can't be used with SET GLOBAL
```

- Use of `SESSION` to set a variable that has only a global value:

```
mysql> SET SESSION max_connections = 1000;
ERROR 1229 (HY000): Variable 'max_connections' is a
GLOBAL variable and should be set with SET GLOBAL
```

- Omission of `GLOBAL`, `PERSIST`, or `PERSIST_ONLY` to set a variable that has only a global value:

```
mysql> SET max_connections = 1000;
ERROR 1229 (HY000): Variable 'max_connections' is a
GLOBAL variable and should be set with SET GLOBAL
```

- Use of `PERSIST` or `PERSIST_ONLY` to set a variable that cannot be persisted:

```
mysql> SET PERSIST port = 3307;
ERROR 1238 (HY000): Variable 'port' is a read only variable
mysql> SET PERSIST_ONLY port = 3307;
ERROR 1238 (HY000): Variable 'port' is a non persistent read only variable
```

- The `@@GLOBAL.`, `@@PERSIST.`, `@@PERSIST_ONLY.`, `@@SESSION.`, and `@@` modifiers apply only to system variables. An error occurs for attempts to apply them to user-defined variables, stored procedure or function parameters, or stored program local variables.
- Not all system variables can be set to `DEFAULT`. In such cases, assigning `DEFAULT` results in an error.
- An error occurs for attempts to assign `DEFAULT` to user-defined variables, stored procedure or function parameters, or stored program local variables.

Multiple Variable Assignment

A `SET` statement can contain multiple variable assignments, separated by commas. This statement assigns values to a user-defined variable and a system variable:

```
SET @x = 1, SESSION sql_mode = '';
```

If you set multiple system variables in a single statement, the most recent `GLOBAL`, `PERSIST`, `PERSIST_ONLY`, or `SESSION` keyword in the statement is used for following assignments that have no keyword specified.

Examples of multiple-variable assignment:

```
SET GLOBAL sort_buffer_size = 1000000, SESSION sort_buffer_size = 1000000;
SET @@GLOBAL.sort_buffer_size = 1000000, @@LOCAL.sort_buffer_size = 1000000;
SET GLOBAL max_connections = 1000, sort_buffer_size = 1000000;
```

The `@@GLOBAL.`, `@@PERSIST.`, `@@PERSIST_ONLY.`, `@@SESSION.`, and `@@` modifiers apply only to the immediately following system variable, not any remaining system variables. This statement sets the `sort_buffer_size` global value to 50000 and the session value to 1000000:

```
SET @@GLOBAL.sort_buffer_size = 50000, sort_buffer_size = 1000000;
```

System Variable References in Expressions

To refer to the value of a system variable in expressions, use one of the `@@`-modifiers (except `@@PERSIST.` and `@@PERSIST_ONLY.`, which are not permitted in expressions). For example, you can retrieve system variable values in a `SELECT` statement like this:

```
SELECT @@GLOBAL.sql_mode, @@SESSION.sql_mode, @@sql_mode;
```

Note

A reference to a system variable in an expression as `@@var_name` (with `@@` rather than `@@GLOBAL.` or `@@SESSION.`) returns the session value if it exists and the global value otherwise. This differs from `SET @@var_name = expr`, which always refers to the session value.

13.7.6.2 SET CHARACTER SET Statement

```
SET {CHARACTER SET | CHARSET}
    {'charset_name' | DEFAULT}
```

This statement maps all strings sent between the server and the current client with the given mapping. `SET CHARACTER SET` sets three session system variables: `character_set_client` and `character_set_results` are set to the given character set, and `character_set_connection` to the value of `character_set_database`. See Section 10.4, “Connection Character Sets and Collations”.

`charset_name` may be quoted or unquoted.

The default character set mapping can be restored by using the value `DEFAULT`. The default depends on the server configuration.

Some character sets cannot be used as the client character set. Attempting to use them with `SET CHARACTER SET` produces an error. See Impermissible Client Character Sets.

13.7.6.3 SET NAMES Statement

```
SET NAMES {'charset_name'
    [COLLATE 'collation_name'] | DEFAULT}
```

This statement sets the three session system variables `character_set_client`, `character_set_connection`, and `character_set_results` to the given character set. Setting `character_set_connection` to `charset_name` also sets `collation_connection` to the default collation for `charset_name`. See Section 10.4, “Connection Character Sets and Collations”.

The optional `COLLATE` clause may be used to specify a collation explicitly. If given, the collation must one of the permitted collations for *`charset_name`*.

`charset_name` and *`collation_name`* may be quoted or unquoted.

The default mapping can be restored by using a value of `DEFAULT`. The default depends on the server configuration.

Some character sets cannot be used as the client character set. Attempting to use them with `SET NAMES` produces an error. See Impermissible Client Character Sets.

13.7.7 SHOW Statements

`SHOW` has many forms that provide information about databases, tables, columns, or status information about the server. This section describes those following:

```
SHOW {BINARY | MASTER} LOGS
SHOW BINLOG EVENTS [IN 'log_name'] [FROM pos] [LIMIT [offset,] row_count]
SHOW {CHARACTER SET | CHARSET} [like_or_where]
SHOW COLLATION [like_or_where]
SHOW [FULL] COLUMNS FROM tbl_name [FROM db_name] [like_or_where]
SHOW CREATE DATABASE db_name
SHOW CREATE EVENT event_name
SHOW CREATE FUNCTION func_name
SHOW CREATE PROCEDURE proc_name
SHOW CREATE TABLE tbl_name
```

```
SHOW CREATE TRIGGER trigger_name
SHOW CREATE VIEW view_name
SHOW DATABASES [like_or_where]
SHOW ENGINE engine_name {STATUS | MUTEX}
SHOW [STORAGE] ENGINES
SHOW ERRORS [LIMIT [offset,] row_count]
SHOW EVENTS
SHOW FUNCTION CODE func_name
SHOW FUNCTION STATUS [like_or_where]
SHOW GRANTS FOR user
SHOW INDEX FROM tbl_name [FROM db_name]
SHOW MASTER STATUS
SHOW OPEN TABLES [FROM db_name] [like_or_where]
SHOW PLUGINS
SHOW PROCEDURE CODE proc_name
SHOW PROCEDURE STATUS [like_or_where]
SHOW PRIVILEGES
SHOW [FULL] PROCESSLIST
SHOW PROFILE [types] [FOR QUERY n] [OFFSET n] [LIMIT n]
SHOW PROFILES
SHOW RELAYLOG EVENTS [IN 'log_name'] [FROM pos] [LIMIT [offset,] row_count]
SHOW {REPLICAS | SLAVE HOSTS}
SHOW {REPLICA | SLAVE} STATUS [FOR CHANNEL channel]
SHOW [GLOBAL | SESSION] STATUS [like_or_where]
SHOW TABLE STATUS [FROM db_name] [like_or_where]
SHOW [FULL] TABLES [FROM db_name] [like_or_where]
SHOW TRIGGERS [FROM db_name] [like_or_where]
SHOW [GLOBAL | SESSION] VARIABLES [like_or_where]
SHOW WARNINGS [LIMIT [offset,] row_count]

like_or_where: {
    LIKE 'pattern'
  | WHERE expr
}
```

If the syntax for a given `SHOW` statement includes a `LIKE 'pattern'` part, `'pattern'` is a string that can contain the SQL `%` and `_` wildcard characters. The pattern is useful for restricting statement output to matching values.

Several `SHOW` statements also accept a `WHERE` clause that provides more flexibility in specifying which rows to display. See Section 26.8, “Extensions to SHOW Statements”.

In `SHOW` statement results, user names and host names are quoted using backticks (`).

Many MySQL APIs (such as PHP) enable you to treat the result returned from a `SHOW` statement as you would a result set from a `SELECT`; see Chapter 29, *Connectors and APIs*, or your API documentation for more information. In addition, you can work in SQL with results from queries on tables in the `INFORMATION_SCHEMA` database, which you cannot easily do with results from `SHOW` statements. See Chapter 26, *INFORMATION_SCHEMA Tables*.

13.7.7.1 SHOW BINARY LOGS Statement

```
SHOW BINARY LOGS
SHOW MASTER LOGS
```

Lists the binary log files on the server. This statement is used as part of the procedure described in Section 13.4.1.1, “PURGE BINARY LOGS Statement”, that shows how to determine which logs can be purged. `SHOW BINARY LOGS` requires the `REPLICATION CLIENT` privilege (or the deprecated `SUPER` privilege).

Encrypted binary log files have a 512-byte file header that stores information required for encryption and decryption of the file. This is included in the file size displayed by `SHOW BINARY LOGS`. The `Encrypted` column shows whether or not the binary log file is encrypted. Binary log encryption is active if `binlog_encryption=ON` is set for the server. Existing binary log files are not encrypted or decrypted if binary log encryption is activated or deactivated while the server is running.

```
mysql> SHOW BINARY LOGS;
```

```
+---------------+-----------+-----------+
| Log_name      | File_size | Encrypted |
+---------------+-----------+-----------+
| binlog.000015 |    724935 |       Yes |
| binlog.000016 |    733481 |       Yes |
+---------------+-----------+-----------+
```

`SHOW MASTER LOGS` is equivalent to `SHOW BINARY LOGS`.

13.7.7.2 SHOW BINLOG EVENTS Statement

```
SHOW BINLOG EVENTS
   [IN 'log_name']
   [FROM pos]
   [LIMIT [offset,] row_count]
```

Shows the events in the binary log. If you do not specify `'log_name'`, the first binary log is displayed. `SHOW BINLOG EVENTS` requires the `REPLICATION SLAVE` privilege.

The `LIMIT` clause has the same syntax as for the `SELECT` statement. See Section 13.2.13, “SELECT Statement”.

Note

Issuing a `SHOW BINLOG EVENTS` with no `LIMIT` clause could start a very time- and resource-consuming process because the server returns to the client the complete contents of the binary log (which includes all statements executed by the server that modify data). As an alternative to `SHOW BINLOG EVENTS`, use the `mysqlbinlog` utility to save the binary log to a text file for later examination and analysis. See Section 4.6.9, “mysqlbinlog — Utility for Processing Binary Log Files”.

`SHOW BINLOG EVENTS` displays the following fields for each event in the binary log:

- `Log_name`

  The name of the file that is being listed.

- `Pos`

  The position at which the event occurs.

- `Event_type`

  An identifier that describes the event type.

- `Server_id`

  The server ID of the server on which the event originated.

- `End_log_pos`

  The position at which the next event begins, which is equal to `Pos` plus the size of the event.

- `Info`

  More detailed information about the event type. The format of this information depends on the event type.

For compressed transaction payloads, the `Transaction_payload_event` is first printed as a single unit, then it is unpacked and each event inside it is printed.

Some events relating to the setting of user and system variables are not included in the output from `SHOW BINLOG EVENTS`. To get complete coverage of events within a binary log, use `mysqlbinlog`.

`SHOW BINLOG EVENTS` does *not* work with relay log files. You can use `SHOW RELAYLOG EVENTS` for this purpose.

13.7.7.3 SHOW CHARACTER SET Statement

```
SHOW {CHARACTER SET | CHARSET}
    [LIKE 'pattern' | WHERE expr]
```

The `SHOW CHARACTER SET` statement shows all available character sets. The `LIKE` clause, if present, indicates which character set names to match. The `WHERE` clause can be given to select rows using more general conditions, as discussed in Section 26.8, “Extensions to SHOW Statements”. For example:

```
mysql> SHOW CHARACTER SET LIKE 'latin%';
+---------+-----------------------------+-------------------+--------+
| Charset | Description                 | Default collation | Maxlen |
+---------+-----------------------------+-------------------+--------+
| latin1  | cp1252 West European        | latin1_swedish_ci |      1 |
| latin2  | ISO 8859-2 Central European | latin2_general_ci |      1 |
| latin5  | ISO 8859-9 Turkish          | latin5_turkish_ci |      1 |
| latin7  | ISO 8859-13 Baltic          | latin7_general_ci |      1 |
+---------+-----------------------------+-------------------+--------+
```

`SHOW CHARACTER SET` output has these columns:

- `Charset`

  The character set name.

- `Description`

  A description of the character set.

- `Default collation`

  The default collation for the character set.

- `Maxlen`

  The maximum number of bytes required to store one character.

The `filename` character set is for internal use only; consequently, `SHOW CHARACTER SET` does not display it.

Character set information is also available from the `INFORMATION_SCHEMA CHARACTER_SETS` table.

13.7.7.4 SHOW COLLATION Statement

```
SHOW COLLATION
    [LIKE 'pattern' | WHERE expr]
```

This statement lists collations supported by the server. By default, the output from `SHOW COLLATION` includes all available collations. The `LIKE` clause, if present, indicates which collation names to match. The `WHERE` clause can be given to select rows using more general conditions, as discussed in Section 26.8, “Extensions to SHOW Statements”. For example:

```
mysql> SHOW COLLATION WHERE Charset = 'latin1';
+-------------------+---------+----+---------+----------+---------+
| Collation         | Charset | Id | Default | Compiled | Sortlen |
+-------------------+---------+----+---------+----------+---------+
| latin1_german1_ci | latin1  |  5 |         | Yes      |       1 |
| latin1_swedish_ci | latin1  |  8 | Yes     | Yes      |       1 |
| latin1_danish_ci  | latin1  | 15 |         | Yes      |       1 |
| latin1_german2_ci | latin1  | 31 |         | Yes      |       2 |
| latin1_bin        | latin1  | 47 |         | Yes      |       1 |
```

```
| latin1_general_ci | latin1  | 48 |         | Yes      |       1 |
| latin1_general_cs | latin1  | 49 |         | Yes      |       1 |
| latin1_spanish_ci | latin1  | 94 |         | Yes      |       1 |
+-------------------+---------+----+---------+----------+---------+
```

`SHOW COLLATION` output has these columns:

- `Collation`

  The collation name.

- `Charset`

  The name of the character set with which the collation is associated.

- `Id`

  The collation ID.

- `Default`

  Whether the collation is the default for its character set.

- `Compiled`

  Whether the character set is compiled into the server.

- `Sortlen`

  This is related to the amount of memory required to sort strings expressed in the character set.

To see the default collation for each character set, use the following statement. `Default` is a reserved word, so to use it as an identifier, it must be quoted as such:

```
mysql> SHOW COLLATION WHERE `Default` = 'Yes';
+---------------------+----------+----+---------+----------+---------+
| Collation           | Charset  | Id | Default | Compiled | Sortlen |
+---------------------+----------+----+---------+----------+---------+
| big5_chinese_ci     | big5     |  1 | Yes     | Yes      |       1 |
| dec8_swedish_ci     | dec8     |  3 | Yes     | Yes      |       1 |
| cp850_general_ci    | cp850    |  4 | Yes     | Yes      |       1 |
| hp8_english_ci      | hp8      |  6 | Yes     | Yes      |       1 |
| koi8r_general_ci    | koi8r    |  7 | Yes     | Yes      |       1 |
| latin1_swedish_ci   | latin1   |  8 | Yes     | Yes      |       1 |
...
```

Collation information is also available from the `INFORMATION_SCHEMA COLLATIONS` table. See Section 26.3.6, “The INFORMATION_SCHEMA COLLATIONS Table”.

13.7.7.5 SHOW COLUMNS Statement

```
SHOW [EXTENDED] [FULL] {COLUMNS | FIELDS}
    {FROM | IN} tbl_name
    [{FROM | IN} db_name]
    [LIKE 'pattern' | WHERE expr]
```

`SHOW COLUMNS` displays information about the columns in a given table. It also works for views. `SHOW COLUMNS` displays information only for those columns for which you have some privilege.

```
mysql> SHOW COLUMNS FROM City;
+-------------+----------+------+-----+---------+----------------+
| Field       | Type     | Null | Key | Default | Extra          |
+-------------+----------+------+-----+---------+----------------+
| ID          | int(11)  | NO   | PRI | NULL    | auto_increment |
| Name        | char(35) | NO   |     |         |                |
| CountryCode | char(3)  | NO   | MUL |         |                |
| District    | char(20) | NO   |     |         |                |
```

```
| Population  | int(11)  | NO   |     | 0       |                |
+-------------+----------+------+-----+---------+----------------+
```

An alternative to *`tbl_name FROM db_name`* syntax is *`db_name.tbl_name`*. These two statements are equivalent:

```
SHOW COLUMNS FROM mytable FROM mydb;
SHOW COLUMNS FROM mydb.mytable;
```

The optional `EXTENDED` keyword causes the output to include information about hidden columns that MySQL uses internally and are not accessible by users.

The optional `FULL` keyword causes the output to include the column collation and comments, as well as the privileges you have for each column.

The `LIKE` clause, if present, indicates which column names to match. The `WHERE` clause can be given to select rows using more general conditions, as discussed in Section 26.8, “Extensions to SHOW Statements”.

The data types may differ from what you expect them to be based on a `CREATE TABLE` statement because MySQL sometimes changes data types when you create or alter a table. The conditions under which this occurs are described in Section 13.1.20.7, “Silent Column Specification Changes”.

`SHOW COLUMNS` displays the following values for each table column:

- `Field`

  The name of the column.

- `Type`

  The column data type.

- `Collation`

  The collation for nonbinary string columns, or `NULL` for other columns. This value is displayed only if you use the `FULL` keyword.

- `Null`

  The column nullability. The value is `YES` if `NULL` values can be stored in the column, `NO` if not.

- `Key`

  Whether the column is indexed:

  - If `Key` is empty, the column either is not indexed or is indexed only as a secondary column in a multiple-column, nonunique index.
  - If `Key` is `PRI`, the column is a `PRIMARY KEY` or is one of the columns in a multiple-column `PRIMARY KEY`.
  - If `Key` is `UNI`, the column is the first column of a `UNIQUE` index. (A `UNIQUE` index permits multiple `NULL` values, but you can tell whether the column permits `NULL` by checking the `Null` field.)
  - If `Key` is `MUL`, the column is the first column of a nonunique index in which multiple occurrences of a given value are permitted within the column.

  If more than one of the `Key` values applies to a given column of a table, `Key` displays the one with the highest priority, in the order `PRI`, `UNI`, `MUL`.

  A `UNIQUE` index may be displayed as `PRI` if it cannot contain `NULL` values and there is no `PRIMARY KEY` in the table. A `UNIQUE` index may display as `MUL` if several columns form a composite `UNIQUE`

index; although the combination of the columns is unique, each column can still hold multiple occurrences of a given value.

- `Default`

  The default value for the column. This is `NULL` if the column has an explicit default of `NULL`, or if the column definition includes no `DEFAULT` clause.

- `Extra`

  Any additional information that is available about a given column. The value is nonempty in these cases:

  - `auto_increment` for columns that have the `AUTO_INCREMENT` attribute.
  - `on update CURRENT_TIMESTAMP` for `TIMESTAMP` or `DATETIME` columns that have the `ON UPDATE CURRENT_TIMESTAMP` attribute.
  - `VIRTUAL GENERATED` or `STORED GENERATED` for generated columns.
  - `DEFAULT_GENERATED` for columns that have an expression default value.

- `Privileges`

  The privileges you have for the column. This value is displayed only if you use the `FULL` keyword.

- `Comment`

  Any comment included in the column definition. This value is displayed only if you use the `FULL` keyword.

Table column information is also available from the `INFORMATION_SCHEMA COLUMNS` table. See Section 26.3.8, “The INFORMATION_SCHEMA COLUMNS Table”. The extended information about hidden columns is available only using `SHOW EXTENDED COLUMNS`; it cannot be obtained from the `COLUMNS` table.

You can list a table's columns with the `mysqlshow *db_name tbl_name*` command.

The `DESCRIBE` statement provides information similar to `SHOW COLUMNS`. See Section 13.8.1, “DESCRIBE Statement”.

The `SHOW CREATE TABLE`, `SHOW TABLE STATUS`, and `SHOW INDEX` statements also provide information about tables. See Section 13.7.7, “SHOW Statements”.

In MySQL 8.0.30 and later, `SHOW COLUMNS` includes the table's generated invisible primary key, if it has one, by default. You can cause this information to be suppressed in the statement's output by setting `show_gipk_in_create_table_and_information_schema = OFF`. For more information, see Section 13.1.20.11, “Generated Invisible Primary Keys”.

13.7.7.6 SHOW CREATE DATABASE Statement

```
SHOW CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
```

Shows the `CREATE DATABASE` statement that creates the named database. If the `SHOW` statement includes an `IF NOT EXISTS` clause, the output too includes such a clause. `SHOW CREATE SCHEMA` is a synonym for `SHOW CREATE DATABASE`.

```
mysql> SHOW CREATE DATABASE test\G
*************************** 1. row ***************************
       Database: test
Create Database: CREATE DATABASE `test` /*!40100 DEFAULT CHARACTER SET utf8mb4
                 COLLATE utf8mb4_0900_ai_ci */ /*!80014 DEFAULT ENCRYPTION='N' */

mysql> SHOW CREATE SCHEMA test\G
```

```
*************************** 1. row ***************************
       Database: test
Create Database: CREATE DATABASE `test` /*!40100 DEFAULT CHARACTER SET utf8mb4
                 COLLATE utf8mb4_0900_ai_ci */ /*!80014 DEFAULT ENCRYPTION='N' */
```

`SHOW CREATE DATABASE` quotes table and column names according to the value of the `sql_quote_show_create` option. See Section 5.1.8, “Server System Variables”.

13.7.7.7 SHOW CREATE EVENT Statement

```
SHOW CREATE EVENT event_name
```

This statement displays the `CREATE EVENT` statement needed to re-create a given event. It requires the `EVENT` privilege for the database from which the event is to be shown. For example (using the same event `e_daily` defined and then altered in Section 13.7.7.18, “SHOW EVENTS Statement”):

```
mysql> SHOW CREATE EVENT myschema.e_daily\G
*************************** 1. row ***************************
               Event: e_daily
            sql_mode: ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,
                      NO_ZERO_IN_DATE,NO_ZERO_DATE,
                      ERROR_FOR_DIVISION_BY_ZERO,
                      NO_ENGINE_SUBSTITUTION
           time_zone: SYSTEM
        Create Event: CREATE DEFINER=`jon`@`ghidora` EVENT `e_daily`
                        ON SCHEDULE EVERY 1 DAY
                        STARTS CURRENT_TIMESTAMP + INTERVAL 6 HOUR
                        ON COMPLETION NOT PRESERVE
                        ENABLE
                        COMMENT 'Saves total number of sessions then
                                clears the table each day'
                        DO BEGIN
                          INSERT INTO site_activity.totals (time, total)
                            SELECT CURRENT_TIMESTAMP, COUNT(*)
                              FROM site_activity.sessions;
                          DELETE FROM site_activity.sessions;
                        END
character_set_client: utf8mb4
collation_connection: utf8mb4_0900_ai_ci
  Database Collation: utf8mb4_0900_ai_ci
```

`character_set_client` is the session value of the `character_set_client` system variable when the event was created. `collation_connection` is the session value of the `collation_connection` system variable when the event was created. `Database Collation` is the collation of the database with which the event is associated.

The output reflects the current status of the event (`ENABLE`) rather than the status with which it was created.

13.7.7.8 SHOW CREATE FUNCTION Statement

```
SHOW CREATE FUNCTION func_name
```

This statement is similar to `SHOW CREATE PROCEDURE` but for stored functions. See Section 13.7.7.9, “SHOW CREATE PROCEDURE Statement”.

13.7.7.9 SHOW CREATE PROCEDURE Statement

```
SHOW CREATE PROCEDURE proc_name
```

This statement is a MySQL extension. It returns the exact string that can be used to re-create the named stored procedure. A similar statement, `SHOW CREATE FUNCTION`, displays information about stored functions (see Section 13.7.7.8, “SHOW CREATE FUNCTION Statement”).

To use either statement, you must be the user named as the routine `DEFINER`, have the `SHOW_ROUTINE` privilege, have the `SELECT` privilege at the global level, or have the `CREATE ROUTINE`, `ALTER ROUTINE`, or `EXECUTE` privilege granted at a scope that includes the routine. The

value displayed for the `Create Procedure` or `Create Function` field is `NULL` if you have only `CREATE ROUTINE`, `ALTER ROUTINE`, or `EXECUTE`.

```
mysql> SHOW CREATE PROCEDURE test.citycount\G
*************************** 1. row ***************************
           Procedure: citycount
            sql_mode: ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,
                      NO_ZERO_IN_DATE,NO_ZERO_DATE,
                      ERROR_FOR_DIVISION_BY_ZERO,
                      NO_ENGINE_SUBSTITUTION
    Create Procedure: CREATE DEFINER=`me`@`localhost`
                      PROCEDURE `citycount`(IN country CHAR(3), OUT cities INT)
                      BEGIN
                        SELECT COUNT(*) INTO cities FROM world.city
                        WHERE CountryCode = country;
                      END
character_set_client: utf8mb4
collation_connection: utf8mb4_0900_ai_ci
  Database Collation: utf8mb4_0900_ai_ci

mysql> SHOW CREATE FUNCTION test.hello\G
*************************** 1. row ***************************
            Function: hello
            sql_mode: ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,
                      NO_ZERO_IN_DATE,NO_ZERO_DATE,
                      ERROR_FOR_DIVISION_BY_ZERO,
                      NO_ENGINE_SUBSTITUTION
     Create Function: CREATE DEFINER=`me`@`localhost`
                      FUNCTION `hello`(s CHAR(20))
                      RETURNS char(50) CHARSET utf8mb4
                      DETERMINISTIC
                      RETURN CONCAT('Hello, ',s,'!')
character_set_client: utf8mb4
collation_connection: utf8mb4_0900_ai_ci
  Database Collation: utf8mb4_0900_ai_ci
```

`character_set_client` is the session value of the `character_set_client` system variable when the routine was created. `collation_connection` is the session value of the `collation_connection` system variable when the routine was created. `Database Collation` is the collation of the database with which the routine is associated.

13.7.7.10 SHOW CREATE TABLE Statement

```
SHOW CREATE TABLE tbl_name
```

Shows the `CREATE TABLE` statement that creates the named table. To use this statement, you must have some privilege for the table. This statement also works with views.

```
mysql> SHOW CREATE TABLE t\G
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `id` int NOT NULL AUTO_INCREMENT,
  `s` char(60) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

As of MySQL 8.0.16, MySQL implements `CHECK` constraints and `SHOW CREATE TABLE` displays them. All `CHECK` constraints are displayed as table constraints. That is, a `CHECK` constraint originally specified as part of a column definition displays as a separate clause not part of the column definition. Example:

```
mysql> CREATE TABLE t1 (
         i1 INT CHECK (i1 <> 0),      -- column constraint
         i2 INT,
         CHECK (i2 > i1),             -- table constraint
         CHECK (i2 <> 0) NOT ENFORCED -- table constraint, not enforced
       );
```

```
mysql> SHOW CREATE TABLE t1\G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `i1` int DEFAULT NULL,
  `i2` int DEFAULT NULL,
  CONSTRAINT `t1_chk_1` CHECK ((`i1` <> 0)),
  CONSTRAINT `t1_chk_2` CHECK ((`i2` > `i1`)),
  CONSTRAINT `t1_chk_3` CHECK ((`i2` <> 0)) /*!80016 NOT ENFORCED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

`SHOW CREATE TABLE` quotes table and column names according to the value of the `sql_quote_show_create` option. See Section 5.1.8, “Server System Variables”.

When altering the storage engine of a table, table options that are not applicable to the new storage engine are retained in the table definition to enable reverting the table with its previously defined options to the original storage engine, if necessary. For example, when changing the storage engine from `InnoDB` to `MyISAM`, options specific to `InnoDB`, such as `ROW_FORMAT=COMPACT`, are retained, as shown here:

```
mysql> CREATE TABLE t1 (c1 INT PRIMARY KEY) ROW_FORMAT=COMPACT ENGINE=InnoDB;
mysql> ALTER TABLE t1 ENGINE=MyISAM;
mysql> SHOW CREATE TABLE t1\G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `c1` int NOT NULL,
  PRIMARY KEY (`c1`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci ROW_FORMAT=COMPACT
```

When creating a table with strict mode disabled, the storage engine's default row format is used if the specified row format is not supported. The actual row format of the table is reported in the `Row_format` column in response to `SHOW TABLE STATUS`. `SHOW CREATE TABLE` shows the row format that was specified in the `CREATE TABLE` statement.

In MySQL 8.0.30 and later, `SHOW CREATE TABLE` includes the definition of the table's generated invisible primary key, if it has such a key, by default. You can cause this information to be suppressed in the statement's output by setting `show_gipk_in_create_table_and_information_schema = OFF`. For more information, see Section 13.1.20.11, “Generated Invisible Primary Keys”.

13.7.7.11 SHOW CREATE TRIGGER Statement

```
SHOW CREATE TRIGGER trigger_name
```

This statement shows the `CREATE TRIGGER` statement that creates the named trigger. This statement requires the `TRIGGER` privilege for the table associated with the trigger.

```
mysql> SHOW CREATE TRIGGER ins_sum\G
*************************** 1. row ***************************
               Trigger: ins_sum
              sql_mode: ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,
                        NO_ZERO_IN_DATE,NO_ZERO_DATE,
                        ERROR_FOR_DIVISION_BY_ZERO,
                        NO_ENGINE_SUBSTITUTION
SQL Original Statement: CREATE DEFINER=`me`@`localhost` TRIGGER `ins_sum`
                        BEFORE INSERT ON `account`
                        FOR EACH ROW SET @sum = @sum + NEW.amount
  character_set_client: utf8mb4
  collation_connection: utf8mb4_0900_ai_ci
    Database Collation: utf8mb4_0900_ai_ci
               Created: 2018-08-08 10:10:12.61
```

`SHOW CREATE TRIGGER` output has these columns:

- `Trigger`: The trigger name.
- `sql_mode`: The SQL mode in effect when the trigger executes.

- `SQL Original Statement`: The `CREATE TRIGGER` statement that defines the trigger.
- `character_set_client`: The session value of the `character_set_client` system variable when the trigger was created.
- `collation_connection`: The session value of the `collation_connection` system variable when the trigger was created.
- `Database Collation`: The collation of the database with which the trigger is associated.
- `Created`: The date and time when the trigger was created. This is a `TIMESTAMP(2)` value (with a fractional part in hundredths of seconds) for triggers.

Trigger information is also available from the `INFORMATION_SCHEMA TRIGGERS` table. See Section 26.3.45, “The INFORMATION_SCHEMA TRIGGERS Table”.

13.7.7.12 SHOW CREATE USER Statement

```
SHOW CREATE USER user
```

This statement shows the `CREATE USER` statement that creates the named user. An error occurs if the user does not exist. The statement requires the `SELECT` privilege for the `mysql` system schema, except to see information for the current user. For the current user, the `SELECT` privilege for the `mysql.user` system table is required for display of the password hash in the `IDENTIFIED AS` clause; otherwise, the hash displays as `<secret>`.

To name the account, use the format described in Section 6.2.4, “Specifying Account Names”. The host name part of the account name, if omitted, defaults to `'%'`. It is also possible to specify `CURRENT_USER` or `CURRENT_USER()` to refer to the account associated with the current session.

Password hash values displayed in the `IDENTIFIED WITH` clause of output from `SHOW CREATE USER` may contain unprintable characters that have adverse effects on terminal displays and in other environments. Enabling the `print_identified_with_as_hex` system variable (available as of MySQL 8.0.17) causes `SHOW CREATE USER` to display such hash values as hexadecimal strings rather than as regular string literals. Hash values that do not contain unprintable characters still display as regular string literals, even with this variable enabled.

```
mysql> CREATE USER 'u1'@'localhost' IDENTIFIED BY 'secret';
mysql> SET print_identified_with_as_hex = ON;
mysql> SHOW CREATE USER 'u1'@'localhost'\G
*************************** 1. row ***************************
CREATE USER for u1@localhost: CREATE USER `u1`@`localhost`
IDENTIFIED WITH 'caching_sha2_password'
AS 0x244124303035240C7745603626313D613C4C10633E0A104B1E14135A544A7871567245614F487234464354633365 46F624F
REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT UNLOCK
PASSWORD HISTORY DEFAULT PASSWORD REUSE INTERVAL DEFAULT
PASSWORD REQUIRE CURRENT DEFAULT
```

To display the privileges granted to an account, use the `SHOW GRANTS` statement. See Section 13.7.7.21, “SHOW GRANTS Statement”.

13.7.7.13 SHOW CREATE VIEW Statement

```
SHOW CREATE VIEW view_name
```

This statement shows the `CREATE VIEW` statement that creates the named view.

```
mysql> SHOW CREATE VIEW v\G
*************************** 1. row ***************************
                View: v
         Create View: CREATE ALGORITHM=UNDEFINED
                      DEFINER=`bob`@`localhost`
                      SQL SECURITY DEFINER VIEW
                      `v` AS select 1 AS `a`,2 AS `b`
character_set_client: utf8mb4
```

```
collation_connection: utf8mb4_0900_ai_ci
```

`character_set_client` is the session value of the `character_set_client` system variable when the view was created. `collation_connection` is the session value of the `collation_connection` system variable when the view was created.

Use of `SHOW CREATE VIEW` requires the `SHOW VIEW` privilege, and the `SELECT` privilege for the view in question.

View information is also available from the `INFORMATION_SCHEMA VIEWS` table. See Section 26.3.48, "The INFORMATION_SCHEMA VIEWS Table".

MySQL lets you use different `sql_mode` settings to tell the server the type of SQL syntax to support. For example, you might use the `ANSI` SQL mode to ensure MySQL correctly interprets the standard SQL concatenation operator, the double bar (`||`), in your queries. If you then create a view that concatenates items, you might worry that changing the `sql_mode` setting to a value different from `ANSI` could cause the view to become invalid. But this is not the case. No matter how you write out a view definition, MySQL always stores it the same way, in a canonical form. Here is an example that shows how the server changes a double bar concatenation operator to a `CONCAT()` function:

```
mysql> SET sql_mode = 'ANSI';
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE VIEW test.v AS SELECT 'a' || 'b' as col1;
Query OK, 0 rows affected (0.01 sec)

mysql> SHOW CREATE VIEW test.v\G
*************************** 1. row ***************************
                View: v
         Create View: CREATE VIEW "v" AS select concat('a','b') AS "col1"
...
1 row in set (0.00 sec)
```

The advantage of storing a view definition in canonical form is that changes made later to the value of `sql_mode` do not affect the results from the view. However an additional consequence is that comments prior to `SELECT` are stripped from the definition by the server.

13.7.7.14 SHOW DATABASES Statement

```
SHOW {DATABASES | SCHEMAS}
    [LIKE 'pattern' | WHERE expr]
```

`SHOW DATABASES` lists the databases on the MySQL server host. `SHOW SCHEMAS` is a synonym for `SHOW DATABASES`. The `LIKE` clause, if present, indicates which database names to match. The `WHERE` clause can be given to select rows using more general conditions, as discussed in Section 26.8, "Extensions to SHOW Statements".

You see only those databases for which you have some kind of privilege, unless you have the global `SHOW DATABASES` privilege. You can also get this list using the `mysqlshow` command.

If the server was started with the `--skip-show-database` option, you cannot use this statement at all unless you have the `SHOW DATABASES` privilege.

MySQL implements databases as directories in the data directory, so this statement simply lists directories in that location. However, the output may include names of directories that do not correspond to actual databases.

Database information is also available from the `INFORMATION_SCHEMA SCHEMATA` table. See Section 26.3.31, "The INFORMATION_SCHEMA SCHEMATA Table".

Caution

Because any static global privilege is considered a privilege for all databases, any static global privilege enables a user to see all database names with `SHOW`

> DATABASES or by examining the SCHEMATA table of INFORMATION_SCHEMA, except databases that have been restricted at the database level by partial revokes.

13.7.7.15 SHOW ENGINE Statement

```
SHOW ENGINE engine_name {STATUS | MUTEX}
```

SHOW ENGINE displays operational information about a storage engine. It requires the PROCESS privilege. The statement has these variants:

```
SHOW ENGINE INNODB STATUS
SHOW ENGINE INNODB MUTEX
SHOW ENGINE PERFORMANCE_SCHEMA STATUS
```

SHOW ENGINE INNODB STATUS displays extensive information from the standard InnoDB Monitor about the state of the InnoDB storage engine. For information about the standard monitor and other InnoDB Monitors that provide information about InnoDB processing, see Section 15.17, “InnoDB Monitors”.

SHOW ENGINE INNODB MUTEX displays InnoDB mutex and rw-lock statistics.

> **Note**
>
> InnoDB mutexes and rwlocks can also be monitored using Performance Schema tables. See Section 15.16.2, “Monitoring InnoDB Mutex Waits Using Performance Schema”.

Mutex statistics collection is configured dynamically using the following options:

- To enable the collection of mutex statistics, run:

  ```
  SET GLOBAL innodb_monitor_enable='latch';
  ```

- To reset mutex statistics, run:

  ```
  SET GLOBAL innodb_monitor_reset='latch';
  ```

- To disable the collection of mutex statistics, run:

  ```
  SET GLOBAL innodb_monitor_disable='latch';
  ```

Collection of mutex statistics for SHOW ENGINE INNODB MUTEX can also be enabled by setting innodb_monitor_enable='all', or disabled by setting innodb_monitor_disable='all'.

SHOW ENGINE INNODB MUTEX output has these columns:

- Type

  Always InnoDB.

- Name

  For mutexes, the Name field reports only the mutex name. For rwlocks, the Name field reports the source file where the rwlock is implemented, and the line number in the file where the rwlock is created. The line number is specific to your version of MySQL.

- Status

  The mutex status. This field reports the number of spins, waits, and calls. Statistics for low-level operating system mutexes, which are implemented outside of InnoDB, are not reported.

  - spins indicates the number of spins.

- `waits` indicates the number of mutex waits.
- `calls` indicates how many times the mutex was requested.

`SHOW ENGINE INNODB MUTEX` does not list mutexes and rw-locks for each buffer pool block, as the amount of output would be overwhelming on systems with a large buffer pool. `SHOW ENGINE INNODB MUTEX` does, however, print aggregate `BUF_BLOCK_MUTEX` spin, wait, and call values for buffer pool block mutexes and rw-locks. `SHOW ENGINE INNODB MUTEX` also does not list any mutexes or rw-locks that have never been waited on (`os_waits=0`). Thus, `SHOW ENGINE INNODB MUTEX` only displays information about mutexes and rw-locks outside of the buffer pool that have caused at least one OS-level wait.

Use `SHOW ENGINE PERFORMANCE_SCHEMA STATUS` to inspect the internal operation of the Performance Schema code:

```
mysql> SHOW ENGINE PERFORMANCE_SCHEMA STATUS\G
...
*************************** 3. row ***************************
  Type: performance_schema
  Name: events_waits_history.size
Status: 76
*************************** 4. row ***************************
  Type: performance_schema
  Name: events_waits_history.count
Status: 10000
*************************** 5. row ***************************
  Type: performance_schema
  Name: events_waits_history.memory
Status: 760000
...
*************************** 57. row ***************************
  Type: performance_schema
  Name: performance_schema.memory
Status: 26459600
...
```

This statement is intended to help the DBA understand the effects that different Performance Schema options have on memory requirements.

`Name` values consist of two parts, which name an internal buffer and a buffer attribute, respectively. Interpret buffer names as follows:

- An internal buffer that is not exposed as a table is named within parentheses. Examples: `(pfs_cond_class).size`, `(pfs_mutex_class).memory`.
- An internal buffer that is exposed as a table in the `performance_schema` database is named after the table, without parentheses. Examples: `events_waits_history.size`, `mutex_instances.count`.
- A value that applies to the Performance Schema as a whole begins with `performance_schema`. Example: `performance_schema.memory`.

Buffer attributes have these meanings:

- `size` is the size of the internal record used by the implementation, such as the size of a row in a table. `size` values cannot be changed.
- `count` is the number of internal records, such as the number of rows in a table. `count` values can be changed using Performance Schema configuration options.
- For a table, *`tbl_name`*`.memory` is the product of `size` and `count`. For the Performance Schema as a whole, `performance_schema.memory` is the sum of all the memory used (the sum of all other `memory` values).

In some cases, there is a direct relationship between a Performance Schema configuration parameter and a `SHOW ENGINE` value. For example, `events_waits_history_long.count` corresponds to `performance_schema_events_waits_history_long_size`. In other cases, the relationship is more complex. For example, `events_waits_history.count` corresponds to `performance_schema_events_waits_history_size` (the number of rows per thread) multiplied by `performance_schema_max_thread_instances` (the number of threads).

SHOW ENGINE NDB STATUS. If the server has the `NDB` storage engine enabled, `SHOW ENGINE NDB STATUS` displays cluster status information such as the number of connected data nodes, the cluster connectstring, and cluster binary log epochs, as well as counts of various Cluster API objects created by the MySQL Server when connected to the cluster. Sample output from this statement is shown here:

```
mysql> SHOW ENGINE NDB STATUS;
+------------+----------------------+--------------------------------------------------+
| Type       | Name                 | Status                                           |
+------------+----------------------+--------------------------------------------------+
| ndbcluster | connection           | cluster_node_id=7,
  connected_host=198.51.100.103, connected_port=1186, number_of_data_nodes=4,
  number_of_ready_data_nodes=3, connect_count=0                                       |
| ndbcluster | NdbTransaction       | created=6, free=0, sizeof=212                    |
| ndbcluster | NdbOperation         | created=8, free=8, sizeof=660                    |
| ndbcluster | NdbIndexScanOperation | created=1, free=1, sizeof=744                   |
| ndbcluster | NdbIndexOperation    | created=0, free=0, sizeof=664                    |
| ndbcluster | NdbRecAttr           | created=1285, free=1285, sizeof=60               |
| ndbcluster | NdbApiSignal         | created=16, free=16, sizeof=136                  |
| ndbcluster | NdbLabel             | created=0, free=0, sizeof=196                    |
| ndbcluster | NdbBranch            | created=0, free=0, sizeof=24                     |
| ndbcluster | NdbSubroutine        | created=0, free=0, sizeof=68                     |
| ndbcluster | NdbCall              | created=0, free=0, sizeof=16                     |
| ndbcluster | NdbBlob              | created=1, free=1, sizeof=264                    |
| ndbcluster | NdbReceiver          | created=4, free=0, sizeof=68                     |
| ndbcluster | binlog               | latest_epoch=155467, latest_trans_epoch=148126,
  latest_received_binlog_epoch=0, latest_handled_binlog_epoch=0,
  latest_applied_binlog_epoch=0                                                       |
+------------+----------------------+--------------------------------------------------+
```

The `Status` column in each of these rows provides information about the MySQL server's connection to the cluster and about the cluster binary log's status, respectively. The `Status` information is in the form of comma-delimited set of name/value pairs.

The `connection` row's `Status` column contains the name/value pairs described in the following table.

Name	Value
`cluster_node_id`	The node ID of the MySQL server in the cluster
`connected_host`	The host name or IP address of the cluster management server to which the MySQL server is connected
`connected_port`	The port used by the MySQL server to connect to the management server (`connected_host`)
`number_of_data_nodes`	The number of data nodes configured for the cluster (that is, the number of `[ndbd]` sections in the cluster `config.ini` file)
`number_of_ready_data_nodes`	The number of data nodes in the cluster that are actually running
`connect_count`	The number of times this `mysqld` has connected or reconnected to cluster data nodes

The `binlog` row's `Status` column contains information relating to NDB Cluster Replication. The name/value pairs it contains are described in the following table.

Name	Value
`latest_epoch`	The most recent epoch most recently run on this MySQL server (that is, the sequence number of the most recent transaction run on the server)
`latest_trans_epoch`	The most recent epoch processed by the cluster's data nodes
`latest_received_binlog_epoch`	The most recent epoch received by the binary log thread
`latest_handled_binlog_epoch`	The most recent epoch processed by the binary log thread (for writing to the binary log)
`latest_applied_binlog_epoch`	The most recent epoch actually written to the binary log

See Section 23.7, “NDB Cluster Replication”, for more information.

The remaining rows from the output of `SHOW ENGINE NDB STATUS` which are most likely to prove useful in monitoring the cluster are listed here by `Name`:

- `NdbTransaction`: The number and size of `NdbTransaction` objects that have been created. An `NdbTransaction` is created each time a table schema operation (such as `CREATE TABLE` or `ALTER TABLE`) is performed on an `NDB` table.
- `NdbOperation`: The number and size of `NdbOperation` objects that have been created.
- `NdbIndexScanOperation`: The number and size of `NdbIndexScanOperation` objects that have been created.
- `NdbIndexOperation`: The number and size of `NdbIndexOperation` objects that have been created.
- `NdbRecAttr`: The number and size of `NdbRecAttr` objects that have been created. In general, one of these is created each time a data manipulation statement is performed by an SQL node.
- `NdbBlob`: The number and size of `NdbBlob` objects that have been created. An `NdbBlob` is created for each new operation involving a `BLOB` column in an `NDB` table.
- `NdbReceiver`: The number and size of any `NdbReceiver` object that have been created. The number in the `created` column is the same as the number of data nodes in the cluster to which the MySQL server has connected.

Note

`SHOW ENGINE NDB STATUS` returns an empty result if no operations involving `NDB` tables have been performed during the current session by the MySQL client accessing the SQL node on which this statement is run.

13.7.7.16 SHOW ENGINES Statement

```
SHOW [STORAGE] ENGINES
```

`SHOW ENGINES` displays status information about the server's storage engines. This is particularly useful for checking whether a storage engine is supported, or to see what the default engine is.

For information about MySQL storage engines, see Chapter 15, *The InnoDB Storage Engine*, and Chapter 16, *Alternative Storage Engines*.

```
mysql> SHOW ENGINES\G
*************************** 1. row ***************************
      Engine: ARCHIVE
```

```
     Support: YES
     Comment: Archive storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 2. row ***************************
      Engine: BLACKHOLE
     Support: YES
     Comment: /dev/null storage engine (anything you write to it disappears)
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 3. row ***************************
      Engine: MRG_MYISAM
     Support: YES
     Comment: Collection of identical MyISAM tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 4. row ***************************
      Engine: FEDERATED
     Support: NO
     Comment: Federated MySQL storage engine
Transactions: NULL
          XA: NULL
  Savepoints: NULL
*************************** 5. row ***************************
      Engine: MyISAM
     Support: YES
     Comment: MyISAM storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 6. row ***************************
      Engine: PERFORMANCE_SCHEMA
     Support: YES
     Comment: Performance Schema
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 7. row ***************************
      Engine: InnoDB
     Support: DEFAULT
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
*************************** 8. row ***************************
      Engine: MEMORY
     Support: YES
     Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 9. row ***************************
      Engine: CSV
     Support: YES
     Comment: CSV storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
```

The output from `SHOW ENGINES` may vary according to the MySQL version used and other factors.

`SHOW ENGINES` output has these columns:

- `Engine`

  The name of the storage engine.

- `Support`

The server's level of support for the storage engine, as shown in the following table.

Value	Meaning
YES	The engine is supported and is active
DEFAULT	Like YES, plus this is the default engine
NO	The engine is not supported
DISABLED	The engine is supported but has been disabled

A value of NO means that the server was compiled without support for the engine, so it cannot be enabled at runtime.

A value of DISABLED occurs either because the server was started with an option that disables the engine, or because not all options required to enable it were given. In the latter case, the error log should contain a reason indicating why the option is disabled. See Section 5.4.2, “The Error Log”.

You might also see DISABLED for a storage engine if the server was compiled to support it, but was started with a `--skip-engine_name` option. For the NDB storage engine, DISABLED means the server was compiled with support for NDB Cluster, but was not started with the `--ndbcluster` option.

All MySQL servers support MyISAM tables. It is not possible to disable MyISAM.

- Comment

  A brief description of the storage engine.

- Transactions

  Whether the storage engine supports transactions.

- XA

  Whether the storage engine supports XA transactions.

- Savepoints

  Whether the storage engine supports savepoints.

Storage engine information is also available from the INFORMATION_SCHEMA ENGINES table. See Section 26.3.13, “The INFORMATION_SCHEMA ENGINES Table”.

13.7.7.17 SHOW ERRORS Statement

```
SHOW ERRORS [LIMIT [offset,] row_count]
SHOW COUNT(*) ERRORS
```

SHOW ERRORS is a diagnostic statement that is similar to SHOW WARNINGS, except that it displays information only for errors, rather than for errors, warnings, and notes.

The LIMIT clause has the same syntax as for the SELECT statement. See Section 13.2.13, “SELECT Statement”.

The SHOW COUNT(*) ERRORS statement displays the number of errors. You can also retrieve this number from the error_count variable:

```
SHOW COUNT(*) ERRORS;
SELECT @@error_count;
```

SHOW ERRORS and error_count apply only to errors, not warnings or notes. In other respects, they are similar to SHOW WARNINGS and warning_count. In particular, SHOW ERRORS cannot display

information for more than `max_error_count` messages, and `error_count` can exceed the value of `max_error_count` if the number of errors exceeds `max_error_count`.

For more information, see Section 13.7.7.42, “SHOW WARNINGS Statement”.

13.7.7.18 SHOW EVENTS Statement

```
SHOW EVENTS
    [{FROM | IN} schema_name]
    [LIKE 'pattern' | WHERE expr]
```

This statement displays information about Event Manager events, which are discussed in Section 25.4, “Using the Event Scheduler”. It requires the `EVENT` privilege for the database from which the events are to be shown.

In its simplest form, `SHOW EVENTS` lists all of the events in the current schema:

```
mysql> SELECT CURRENT_USER(), SCHEMA();
+----------------+----------+
| CURRENT_USER() | SCHEMA() |
+----------------+----------+
| jon@ghidora    | myschema |
+----------------+----------+
1 row in set (0.00 sec)

mysql> SHOW EVENTS\G
*************************** 1. row ***************************
                  Db: myschema
                Name: e_daily
             Definer: jon@ghidora
           Time zone: SYSTEM
                Type: RECURRING
          Execute at: NULL
      Interval value: 1
      Interval field: DAY
              Starts: 2018-08-08 11:06:34
                Ends: NULL
              Status: ENABLED
          Originator: 1
character_set_client: utf8mb4
collation_connection: utf8mb4_0900_ai_ci
  Database Collation: utf8mb4_0900_ai_ci
```

To see events for a specific schema, use the `FROM` clause. For example, to see events for the `test` schema, use the following statement:

```
SHOW EVENTS FROM test;
```

The `LIKE` clause, if present, indicates which event names to match. The `WHERE` clause can be given to select rows using more general conditions, as discussed in Section 26.8, “Extensions to SHOW Statements”.

`SHOW EVENTS` output has these columns:

- `Db`

  The name of the schema (database) to which the event belongs.

- `Name`

  The name of the event.

- `Definer`

  The account of the user who created the event, in `'user_name'@'host_name'` format.

- `Time zone`

The event time zone, which is the time zone used for scheduling the event and that is in effect within the event as it executes. The default value is `SYSTEM`.

- `Type`

  The event repetition type, either `ONE TIME` (transient) or `RECURRING` (repeating).

- `Execute At`

  For a one-time event, this is the `DATETIME` value specified in the `AT` clause of the `CREATE EVENT` statement used to create the event, or of the last `ALTER EVENT` statement that modified the event. The value shown in this column reflects the addition or subtraction of any `INTERVAL` value included in the event's `AT` clause. For example, if an event is created using `ON SCHEDULE AT CURRENT_TIMESTAMP + '1:6' DAY_HOUR`, and the event was created at 2018-02-09 14:05:30, the value shown in this column would be `'2018-02-10 20:05:30'`. If the event's timing is determined by an `EVERY` clause instead of an `AT` clause (that is, if the event is recurring), the value of this column is `NULL`.

- `Interval Value`

  For a recurring event, the number of intervals to wait between event executions. For a transient event, the value of this column is always `NULL`.

- `Interval Field`

  The time units used for the interval which a recurring event waits before repeating. For a transient event, the value of this column is always `NULL`.

- `Starts`

  The start date and time for a recurring event. This is displayed as a `DATETIME` value, and is `NULL` if no start date and time are defined for the event. For a transient event, this column is always `NULL`. For a recurring event whose definition includes a `STARTS` clause, this column contains the corresponding `DATETIME` value. As with the `Execute At` column, this value resolves any expressions used. If there is no `STARTS` clause affecting the timing of the event, this column is `NULL`

- `Ends`

  For a recurring event whose definition includes a `ENDS` clause, this column contains the corresponding `DATETIME` value. As with the `Execute At` column, this value resolves any expressions used. If there is no `ENDS` clause affecting the timing of the event, this column is `NULL`.

- `Status`

  The event status. One of `ENABLED`, `DISABLED`, or `SLAVESIDE_DISABLED`. `SLAVESIDE_DISABLED` indicates that the creation of the event occurred on another MySQL server acting as a replication source and replicated to the current MySQL server which is acting as a replica, but the event is not presently being executed on the replica. For more information, see Section 17.5.1.16, “Replication of Invoked Features”. information.

- `Originator`

  The server ID of the MySQL server on which the event was created; used in replication. This value may be updated by `ALTER EVENT` to the server ID of the server on which that statement occurs, if executed on a source server. The default value is 0.

- `character_set_client`

  The session value of the `character_set_client` system variable when the event was created.

- `collation_connection`

The session value of the `collation_connection` system variable when the event was created.

- `Database Collation`

  The collation of the database with which the event is associated.

For more information about `SLAVESIDE_DISABLED` and the `Originator` column, see Section 17.5.1.16, “Replication of Invoked Features”.

Times displayed by `SHOW EVENTS` are given in the event time zone, as discussed in Section 25.4.4, “Event Metadata”.

Event information is also available from the `INFORMATION_SCHEMA EVENTS` table. See Section 26.3.14, “The INFORMATION_SCHEMA EVENTS Table”.

The event action statement is not shown in the output of `SHOW EVENTS`. Use `SHOW CREATE EVENT` or the `INFORMATION_SCHEMA EVENTS` table.

13.7.7.19 SHOW FUNCTION CODE Statement

```
SHOW FUNCTION CODE func_name
```

This statement is similar to `SHOW PROCEDURE CODE` but for stored functions. See Section 13.7.7.27, “SHOW PROCEDURE CODE Statement”.

13.7.7.20 SHOW FUNCTION STATUS Statement

```
SHOW FUNCTION STATUS
    [LIKE 'pattern' | WHERE expr]
```

This statement is similar to `SHOW PROCEDURE STATUS` but for stored functions. See Section 13.7.7.28, “SHOW PROCEDURE STATUS Statement”.

13.7.7.21 SHOW GRANTS Statement

```
SHOW GRANTS
    [FOR user_or_role
        [USING role [, role] ...]]

user_or_role: {
    user (see Section 6.2.4, “Specifying Account Names”)
  | role (see Section 6.2.5, “Specifying Role Names”.
}
```

This statement displays the privileges and roles that are assigned to a MySQL user account or role, in the form of `GRANT` statements that must be executed to duplicate the privilege and role assignments.

Note

To display nonprivilege information for MySQL accounts, use the `SHOW CREATE USER` statement. See Section 13.7.7.12, “SHOW CREATE USER Statement”.

`SHOW GRANTS` requires the `SELECT` privilege for the `mysql` system schema, except to display privileges and roles for the current user.

To name the account or role for `SHOW GRANTS`, use the same format as for the `GRANT` statement (for example, `'jeffrey'@'localhost'`):

```
mysql> SHOW GRANTS FOR 'jeffrey'@'localhost';
+-----------------------------------------------------------------+
| Grants for jeffrey@localhost                                    |
+-----------------------------------------------------------------+
| GRANT USAGE ON *.* TO `jeffrey`@`localhost`                     |
| GRANT SELECT, INSERT, UPDATE ON `db1`.* TO `jeffrey`@`localhost` |
+-----------------------------------------------------------------+
```

The host part, if omitted, defaults to `'%'`. For additional information about specifying account and role names, see Section 6.2.4, “Specifying Account Names”, and Section 6.2.5, “Specifying Role Names”.

To display the privileges granted to the current user (the account you are using to connect to the server), you can use any of the following statements:

```
SHOW GRANTS;
SHOW GRANTS FOR CURRENT_USER;
SHOW GRANTS FOR CURRENT_USER();
```

If `SHOW GRANTS FOR CURRENT_USER` (or any equivalent syntax) is used in definer context, such as within a stored procedure that executes with definer rather than invoker privileges, the grants displayed are those of the definer and not the invoker.

In MySQL 8.0 compared to previous series, `SHOW GRANTS` no longer displays `ALL PRIVILEGES` in its global-privileges output because the meaning of `ALL PRIVILEGES` at the global level varies depending on which dynamic privileges are defined. Instead, `SHOW GRANTS` explicitly lists each granted global privilege:

```
mysql> SHOW GRANTS FOR 'root'@'localhost';
+---------------------------------------------------------------------+
| Grants for root@localhost                                           |
+---------------------------------------------------------------------+
| GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, RELOAD,         |
| SHUTDOWN, PROCESS, FILE, REFERENCES, INDEX, ALTER, SHOW DATABASES,  |
| SUPER, CREATE TEMPORARY TABLES, LOCK TABLES, EXECUTE, REPLICATION   |
| SLAVE, REPLICATION CLIENT, CREATE VIEW, SHOW VIEW, CREATE ROUTINE,  |
| ALTER ROUTINE, CREATE USER, EVENT, TRIGGER, CREATE TABLESPACE,      |
| CREATE ROLE, DROP ROLE ON *.* TO `root`@`localhost` WITH GRANT      |
| OPTION                                                              |
| GRANT PROXY ON ''@'' TO `root`@`localhost` WITH GRANT OPTION        |
+---------------------------------------------------------------------+
```

Applications that process `SHOW GRANTS` output should be adjusted accordingly.

At the global level, `GRANT OPTION` applies to all granted static global privileges if granted for any of them, but applies individually to granted dynamic privileges. `SHOW GRANTS` displays global privileges this way:

- One line listing all granted static privileges, if there are any, including `WITH GRANT OPTION` if appropriate.
- One line listing all granted dynamic privileges for which `GRANT OPTION` is granted, if there are any, including `WITH GRANT OPTION`.
- One line listing all granted dynamic privileges for which `GRANT OPTION` is not granted, if there are any, without `WITH GRANT OPTION`.

With the optional `USING` clause, `SHOW GRANTS` enables you to examine the privileges associated with roles for the user. Each role named in the `USING` clause must be granted to the user.

Suppose that user `u1` is assigned roles `r1` and `r2`, as follows:

```
CREATE ROLE 'r1', 'r2';
GRANT SELECT ON db1.* TO 'r1';
GRANT INSERT, UPDATE, DELETE ON db1.* TO 'r2';
CREATE USER 'u1'@'localhost' IDENTIFIED BY 'u1pass';
GRANT 'r1', 'r2' TO 'u1'@'localhost';
```

`SHOW GRANTS` without `USING` shows the granted roles:

```
mysql> SHOW GRANTS FOR 'u1'@'localhost';
+-----------------------------------------+
| Grants for u1@localhost                 |
+-----------------------------------------+
| GRANT USAGE ON *.* TO `u1`@`localhost`  |
| GRANT `r1`@`%`,`r2`@`%` TO `u1`@`localhost` |
```

```
+-------------------------------------------+
```

Adding a `USING` clause causes the statement to also display the privileges associated with each role named in the clause:

```
mysql> SHOW GRANTS FOR 'u1'@'localhost' USING 'r1';
+-------------------------------------------+
| Grants for u1@localhost                   |
+-------------------------------------------+
| GRANT USAGE ON *.* TO `u1`@`localhost`    |
| GRANT SELECT ON `db1`.* TO `u1`@`localhost` |
| GRANT `r1`@`%`,`r2`@`%` TO `u1`@`localhost` |
+-------------------------------------------+
mysql> SHOW GRANTS FOR 'u1'@'localhost' USING 'r2';
+-----------------------------------------------------------+
| Grants for u1@localhost                                   |
+-----------------------------------------------------------+
| GRANT USAGE ON *.* TO `u1`@`localhost`                    |
| GRANT INSERT, UPDATE, DELETE ON `db1`.* TO `u1`@`localhost` |
| GRANT `r1`@`%`,`r2`@`%` TO `u1`@`localhost`               |
+-----------------------------------------------------------+
mysql> SHOW GRANTS FOR 'u1'@'localhost' USING 'r1', 'r2';
+-----------------------------------------------------------------------+
| Grants for u1@localhost                                               |
+-----------------------------------------------------------------------+
| GRANT USAGE ON *.* TO `u1`@`localhost`                                |
| GRANT SELECT, INSERT, UPDATE, DELETE ON `db1`.* TO `u1`@`localhost` |
| GRANT `r1`@`%`,`r2`@`%` TO `u1`@`localhost`                           |
+-----------------------------------------------------------------------+
```

Note

A privilege granted to an account is always in effect, but a role is not. The active roles for an account can differ across and within sessions, depending on the value of the `activate_all_roles_on_login` system variable, the account default roles, and whether `SET ROLE` has been executed within a session.

MySQL 8.0.16 and higher supports partial revokes of global privileges, such that a global privilege can be restricted from applying to particular schemas (see Section 6.2.12, “Privilege Restriction Using Partial Revokes”). To indicate which global schema privileges have been revoked for particular schemas, `SHOW GRANTS` output includes `REVOKE` statements:

```
mysql> SET PERSIST partial_revokes = ON;
mysql> CREATE USER u1;
mysql> GRANT SELECT, INSERT, DELETE ON *.* TO u1;
mysql> REVOKE SELECT, INSERT ON mysql.* FROM u1;
mysql> REVOKE DELETE ON world.* FROM u1;
mysql> SHOW GRANTS FOR u1;
+-------------------------------------------------+
| Grants for u1@%                                 |
+-------------------------------------------------+
| GRANT SELECT, INSERT, DELETE ON *.* TO `u1`@`%`  |
| REVOKE SELECT, INSERT ON `mysql`.* FROM `u1`@`%` |
| REVOKE DELETE ON `world`.* FROM `u1`@`%`         |
+-------------------------------------------------+
```

`SHOW GRANTS` does not display privileges that are available to the named account but are granted to a different account. For example, if an anonymous account exists, the named account might be able to use its privileges, but `SHOW GRANTS` does not display them.

`SHOW GRANTS` displays mandatory roles named in the `mandatory_roles` system variable value as follows:

- `SHOW GRANTS` without a `FOR` clause displays privileges for the current user, and includes mandatory roles.
- `SHOW GRANTS FOR` *`user`* displays privileges for the named user, and does not include mandatory roles.

This behavior is for the benefit of applications that use the output of `SHOW GRANTS FOR user` to determine which privileges are granted explicitly to the named user. Were that output to include mandatory roles, it would be difficult to distinguish roles granted explicitly to the user from mandatory roles.

For the current user, applications can determine privileges with or without mandatory roles by using `SHOW GRANTS` or `SHOW GRANTS FOR CURRENT_USER`, respectively.

13.7.7.22 SHOW INDEX Statement

```
SHOW [EXTENDED] {INDEX | INDEXES | KEYS}
    {FROM | IN} tbl_name
    [{FROM | IN} db_name]
    [WHERE expr]
```

`SHOW INDEX` returns table index information. The format resembles that of the `SQLStatistics` call in ODBC. This statement requires some privilege for any column in the table.

```
mysql> SHOW INDEX FROM City\G
*************************** 1. row ***************************
        Table: city
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: ID
    Collation: A
  Cardinality: 4188
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
      Visible: YES
   Expression: NULL
*************************** 2. row ***************************
        Table: city
   Non_unique: 1
     Key_name: CountryCode
 Seq_in_index: 1
  Column_name: CountryCode
    Collation: A
  Cardinality: 232
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
      Visible: YES
   Expression: NULL
```

An alternative to *`tbl_name FROM db_name`* syntax is *`db_name.tbl_name`*. These two statements are equivalent:

```
SHOW INDEX FROM mytable FROM mydb;
SHOW INDEX FROM mydb.mytable;
```

The optional `EXTENDED` keyword causes the output to include information about hidden indexes that MySQL uses internally and are not accessible by users.

The `WHERE` clause can be given to select rows using more general conditions, as discussed in Section 26.8, “Extensions to SHOW Statements”.

`SHOW INDEX` returns the following fields:

- `Table`

The name of the table.

- `Non_unique`

  0 if the index cannot contain duplicates, 1 if it can.

- `Key_name`

  The name of the index. If the index is the primary key, the name is always `PRIMARY`.

- `Seq_in_index`

  The column sequence number in the index, starting with 1.

- `Column_name`

  The column name. See also the description for the `Expression` column.

- `Collation`

  How the column is sorted in the index. This can have values `A` (ascending), `D` (descending), or `NULL` (not sorted).

- `Cardinality`

  An estimate of the number of unique values in the index. To update this number, run `ANALYZE TABLE` or (for `MyISAM` tables) `myisamchk -a`.

  `Cardinality` is counted based on statistics stored as integers, so the value is not necessarily exact even for small tables. The higher the cardinality, the greater the chance that MySQL uses the index when doing joins.

- `Sub_part`

  The index prefix. That is, the number of indexed characters if the column is only partly indexed, `NULL` if the entire column is indexed.

  

  Note

  Prefix *limits* are measured in bytes. However, prefix *lengths* for index specifications in `CREATE TABLE`, `ALTER TABLE`, and `CREATE INDEX` statements are 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.

  For additional information about index prefixes, see Section 8.3.5, “Column Indexes”, and Section 13.1.15, “CREATE INDEX Statement”.

- `Packed`

  Indicates how the key is packed. `NULL` if it is not.

- `Null`

  Contains `YES` if the column may contain `NULL` values and `''` if not.

- `Index_type`

  The index method used (`BTREE`, `FULLTEXT`, `HASH`, `RTREE`).

- `Comment`

Information about the index not described in its own column, such as `disabled` if the index is disabled.

- `Index_comment`

  Any comment provided for the index with a `COMMENT` attribute when the index was created.

- `Visible`

  Whether the index is visible to the optimizer. See Section 8.3.12, “Invisible Indexes”.

- `Expression`

  MySQL 8.0.13 and higher supports functional key parts (see Functional Key Parts), which affects both the `Column_name` and `Expression` columns:

  - For a nonfunctional key part, `Column_name` indicates the column indexed by the key part and `Expression` is `NULL`.
  - For a functional key part, `Column_name` column is `NULL` and `Expression` indicates the expression for the key part.

Information about table indexes is also available from the `INFORMATION_SCHEMA STATISTICS` table. See Section 26.3.34, “The INFORMATION_SCHEMA STATISTICS Table”. The extended information about hidden indexes is available only using `SHOW EXTENDED INDEX`; it cannot be obtained from the `STATISTICS` table.

You can list a table's indexes with the `mysqlshow -k db_name tbl_name` command.

In MySQL 8.0.30 and later, `SHOW INDEX` includes the table's generated invisible key, if it has one, by default. You can cause this information to be suppressed in the statement's output by setting `show_gipk_in_create_table_and_information_schema = OFF`. For more information, see Section 13.1.20.11, “Generated Invisible Primary Keys”.

13.7.7.23 SHOW MASTER STATUS Statement

```
SHOW MASTER STATUS
```

This statement provides status information about the binary log files of the source server. It requires the `REPLICATION CLIENT` privilege (or the deprecated `SUPER` privilege).

Example:

```
mysql> SHOW MASTER STATUS\G
*************************** 1. row ***************************
             File: source-bin.000002
         Position: 1307
     Binlog_Do_DB: test
 Binlog_Ignore_DB: manual, mysql
Executed_Gtid_Set: 3E11FA47-71CA-11E1-9E33-C80AA9429562:1-5
1 row in set (0.00 sec)
```

When global transaction IDs are in use, `Executed_Gtid_Set` shows the set of GTIDs for transactions that have been executed on the source. This is the same as the value for the `gtid_executed` system variable on this server, as well as the value for `Executed_Gtid_Set` in the output of `SHOW REPLICA STATUS` (or before MySQL 8.0.22, `SHOW SLAVE STATUS`) on this server.

13.7.7.24 SHOW OPEN TABLES Statement

```
SHOW OPEN TABLES
    [{FROM | IN} db_name]
```

```
    [LIKE 'pattern' | WHERE expr]
```

`SHOW OPEN TABLES` lists the non-`TEMPORARY` tables that are currently open in the table cache. See Section 8.4.3.1, "How MySQL Opens and Closes Tables". The `FROM` clause, if present, restricts the tables shown to those present in the *`db_name`* database. The `LIKE` clause, if present, indicates which table names to match. The `WHERE` clause can be given to select rows using more general conditions, as discussed in Section 26.8, "Extensions to SHOW Statements".

`SHOW OPEN TABLES` output has these columns:

- `Database`

  The database containing the table.

- `Table`

  The table name.

- `In_use`

  The number of table locks or lock requests there are for the table. For example, if one client acquires a lock for a table using `LOCK TABLE t1 WRITE`, `In_use` is 1. If another client issues `LOCK TABLE t1 WRITE` while the table remains locked, the client blocks, waiting for the lock, but the lock request causes `In_use` to be 2. If the count is zero, the table is open but not currently being used. `In_use` is also increased by the `HANDLER ... OPEN` statement and decreased by `HANDLER ... CLOSE`.

- `Name_locked`

  Whether the table name is locked. Name locking is used for operations such as dropping or renaming tables.

If you have no privileges for a table, it does not show up in the output from `SHOW OPEN TABLES`.

13.7.7.25 SHOW PLUGINS Statement

```
SHOW PLUGINS
```

`SHOW PLUGINS` displays information about server plugins.

Example of `SHOW PLUGINS` output:

```
mysql> SHOW PLUGINS\G
*************************** 1. row ***************************
   Name: binlog
 Status: ACTIVE
   Type: STORAGE ENGINE
Library: NULL
License: GPL
*************************** 2. row ***************************
   Name: CSV
 Status: ACTIVE
   Type: STORAGE ENGINE
Library: NULL
License: GPL
*************************** 3. row ***************************
   Name: MEMORY
 Status: ACTIVE
   Type: STORAGE ENGINE
Library: NULL
License: GPL
*************************** 4. row ***************************
   Name: MyISAM
 Status: ACTIVE
   Type: STORAGE ENGINE
Library: NULL
License: GPL
```

```
...
```

`SHOW PLUGINS` output has these columns:

- `Name`

  The name used to refer to the plugin in statements such as `INSTALL PLUGIN` and `UNINSTALL PLUGIN`.

- `Status`

  The plugin status, one of `ACTIVE`, `INACTIVE`, `DISABLED`, `DELETING`, or `DELETED`.

- `Type`

  The type of plugin, such as `STORAGE ENGINE`, `INFORMATION_SCHEMA`, or `AUTHENTICATION`.

- `Library`

  The name of the plugin shared library file. This is the name used to refer to the plugin file in statements such as `INSTALL PLUGIN` and `UNINSTALL PLUGIN`. This file is located in the directory named by the `plugin_dir` system variable. If the library name is `NULL`, the plugin is compiled in and cannot be uninstalled with `UNINSTALL PLUGIN`.

- `License`

  How the plugin is licensed (for example, `GPL`).

For plugins installed with `INSTALL PLUGIN`, the `Name` and `Library` values are also registered in the `mysql.plugin` system table.

For information about plugin data structures that form the basis of the information displayed by `SHOW PLUGINS`, see The MySQL Plugin API.

Plugin information is also available from the `INFORMATION_SCHEMA.PLUGINS` table. See Section 26.3.22, “The INFORMATION_SCHEMA PLUGINS Table”.

13.7.7.26 SHOW PRIVILEGES Statement

```
SHOW PRIVILEGES
```

`SHOW PRIVILEGES` shows the list of system privileges that the MySQL server supports. The privileges displayed include all static privileges, and all currently registered dynamic privileges.

```
mysql> SHOW PRIVILEGES\G
*************************** 1. row ***************************
Privilege: Alter
  Context: Tables
  Comment: To alter the table
*************************** 2. row ***************************
Privilege: Alter routine
  Context: Functions,Procedures
  Comment: To alter or drop stored functions/procedures
*************************** 3. row ***************************
Privilege: Create
  Context: Databases,Tables,Indexes
  Comment: To create new databases and tables
*************************** 4. row ***************************
Privilege: Create routine
  Context: Databases
  Comment: To use CREATE FUNCTION/PROCEDURE
*************************** 5. row ***************************
Privilege: Create temporary tables
  Context: Databases
  Comment: To use CREATE TEMPORARY TABLE
```