MariaDB Server Documentation

MariaDB Knowledge Base

For any errors please see Bug Reporting

Document generated on: 2022-03-18

Chapter Contents

Table of Contents

Chapter 1 SQL Statements and Structure

1.1 SQL Statements

1.1.1 Account Management SQL Commands

CREATE/DROP USER, GRANT, REVOKE, SET PASSWORD etc.

CREATE USER

Create new MariaDB accounts.

ALTER USER

Modify an existing MariaDB account.

DROP USER

Remove one or more MariaDB accounts.

GRANT

Create accounts and set privileges or roles.

RENAME USER

Rename user account.

REVOKE

Remove privileges or roles.

SET PASSWORD

Assign password to an existing MariaDB user.

CREATE ROLE

Add new roles.

DROP ROLE

Drop a role.

SET ROLE

Enable a role.

SET DEFAULT ROLE

Sets a default role for a specified (or current) user.

SHOW GRANTS

View GRANT statements.

SHOW CREATE USER

Show the CREATE USER statement for a specified user.

There are 2 related questions.

1.1.1.1 CREATE USER

Syntax

```
CREATE [OR REPLACE] USER [IF NOT EXISTS]
 user_specification [,user_specification ...]
  [REQUIRE {NONE | tls_option [[AND] tls_option ...] }]
  [WITH resource_option [resource_option ...] ]
  [lock_option] [password_option]

user_specification:
  username [authentication_option]

authentication_option:
  IDENTIFIED BY 'password'
  | IDENTIFIED BY PASSWORD 'password_hash'
  | IDENTIFIED {VIA|WITH} authentication_rule [OR authentication_rule  ...]

authentication_rule:
    authentication_plugin
  | authentication_plugin {USING|AS} 'authentication_string'
  | authentication_plugin {USING|AS} PASSWORD('password')

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
  | MAX_STATEMENT_TIME time

password_option:
  PASSWORD EXPIRE
  | PASSWORD EXPIRE DEFAULT
  | PASSWORD EXPIRE NEVER
  | PASSWORD EXPIRE INTERVAL N DAY

lock_option:
    ACCOUNT LOCK
  | ACCOUNT UNLOCK
}
```

Contents

Description

The `CREATE USER` statement creates new MariaDB accounts. To use it, you must have the global CREATE USER privilege or the INSERT privilege for the mysql database. For each account, `CREATE USER` creates a new row in mysql.user (until MariaDB 10.3 this is a table, from MariaDB 10.4 it's a view) or mysql.global_priv_table (from MariaDB 10.4) that has no privileges.

If any of the specified accounts, or any permissions for the specified accounts, already exist, then the server returns `ERROR 1396 (HY000)`. If an error occurs, `CREATE USER` will still create the accounts that do not result in an error. Only one error is produced for all users which have not been created:

```
ERROR 1396 (HY000):
  Operation CREATE USER failed for 'u1'@'%','u2'@'%'
```

`CREATE USER`, DROP USER, CREATE ROLE, and DROP ROLE all produce the same error code when they fail.

See Account Names below for details on how account names are specified.

OR REPLACE

If the optional `OR REPLACE` clause is used, it is basically a shortcut for:

```
DROP USER IF EXISTS name;
CREATE USER name ...;
```

For example:

```
CREATE USER foo2@test IDENTIFIED BY 'password';
ERROR 1396 (HY000): Operation CREATE USER failed for 'foo2'@'test'

CREATE OR REPLACE USER foo2@test IDENTIFIED BY 'password';
Query OK, 0 rows affected (0.00 sec)
```

IF NOT EXISTS

When the `IF NOT EXISTS` clause is used, MariaDB will return a warning instead of an error if the specified user already exists.

For example:

```
CREATE USER foo2@test IDENTIFIED BY 'password';
ERROR 1396 (HY000): Operation CREATE USER failed for 'foo2'@'test'

CREATE USER IF NOT EXISTS foo2@test IDENTIFIED BY 'password';
Query OK, 0 rows affected, 1 warning (0.00 sec)

SHOW WARNINGS;
+-------+------+-----------------------------------------------------+
| Level | Code | Message                                             |
+-------+------+-----------------------------------------------------+
| Note  | 1973 | Can't create user 'foo2'@'test'; it already exists |
+-------+------+-----------------------------------------------------+
```

Authentication Options

IDENTIFIED BY 'password'

The optional `IDENTIFIED BY` clause can be used to provide an account with a password. The password should be specified in plain text. It will be hashed by the PASSWORD function prior to being stored in the mysql.user/mysql.global_priv_table table.

For example, if our password is `mariadb` , then we can create the user with:

```
CREATE USER foo2@test IDENTIFIED BY 'mariadb';
```

If you do not specify a password with the `IDENTIFIED BY` clause, the user will be able to connect without a password. A blank password is not a wildcard to match any password. The user must connect without providing a password if no password is set.

The only authentication plugins that this clause supports are mysql_native_password and mysql_old_password.

IDENTIFIED BY PASSWORD 'password_hash'

The optional `IDENTIFIED BY PASSWORD` clause can be used to provide an account with a password that has already been hashed. The password should be specified as a hash that was provided by the PASSWORD function. It will be stored in the mysql.user/mysql.global_priv_table table as-is.

For example, if our password is `mariadb` , then we can find the hash with:

```
SELECT PASSWORD('mariadb');
+-------------------------------------------+
| PASSWORD('mariadb')                       |
+-------------------------------------------+
| *54958E764CE10E50764C2EECBB71D01F08549980 |
+-------------------------------------------+
1 row in set (0.00 sec)
```

And then we can create a user with the hash:

```
CREATE USER foo2@test IDENTIFIED BY PASSWORD '*54958E764CE10E50764C2EECBB71D01F08549980';
```

If you do not specify a password with the `IDENTIFIED BY` clause, the user will be able to connect without a password. A blank password is not a wildcard to match any password. The user must connect without providing a password if no password is set.

The only authentication plugins that this clause supports are mysql_native_password and mysql_old_password.

IDENTIFIED {VIA|WITH} authentication_plugin

The optional `IDENTIFIED VIA authentication_plugin` allows you to specify that the account should be authenticated by a specific authentication plugin. The plugin name must be an active authentication plugin as per SHOW PLUGINS. If it doesn't show up in that output, then you will need to install it with INSTALL PLUGIN or INSTALL SONAME.

For example, this could be used with the PAM authentication plugin:

```
CREATE USER foo2@test IDENTIFIED VIA pam;
```

Some authentication plugins allow additional arguments to be specified after a `USING` or `AS` keyword. For example, the PAM authentication plugin accepts a service name:

```
CREATE USER foo2@test IDENTIFIED VIA pam USING 'mariadb';
```

The exact meaning of the additional argument would depend on the specific authentication plugin.

MariaDB starting with 10.4.0

The `USING` or `AS` keyword can also be used to provide a plain-text password to a plugin if it's provided as an argument to the PASSWORD() function. This is only valid for authentication plugins that have implemented a hook for the PASSWORD() function. For example, the ed25519 authentication plugin supports this:

```
CREATE USER safe@'%' IDENTIFIED VIA ed25519 USING PASSWORD('secret');
```

MariaDB starting with 10.4.3

One can specify many authentication plugins, they all work as alternatives ways of authenticating a user:

```
CREATE USER safe@'%' IDENTIFIED VIA ed25519 USING PASSWORD('secret') OR unix_socket;
```

By default, when you create a user without specifying an authentication plugin, MariaDB uses the mysql_native_password plugin.

TLS Options

By default, MariaDB transmits data between the server and clients without encrypting it. This is generally acceptable when the server and client run on the same host or in networks where security is guaranteed through other means. However, in cases where the server and client exist on separate networks or they are in a high-risk network, the lack of encryption does introduce security concerns as a malicious actor could potentially eavesdrop on the traffic as it is sent over the network between them.

To mitigate this concern, MariaDB allows you to encrypt data in transit between the server and clients using the Transport Layer Security (TLS) protocol. TLS was formerly known as Secure Socket Layer (SSL), but strictly speaking the SSL protocol is a predecessor to TLS and, that version of the protocol is now considered insecure. The documentation still uses the term SSL often and for compatibility reasons TLS-related server system and status variables still use the prefix ssl_, but internally, MariaDB only supports its secure successors.

See Secure Connections Overview for more information about how to determine whether your MariaDB server has TLS support.

You can set certain TLS-related restrictions for specific user accounts. For instance, you might use this with user accounts that require access to sensitive data while sending it across networks that you do not control. These restrictions can be enabled for a user account with the CREATE USER, ALTER USER, or GRANT statements. The following options are available:

Option	Description
`REQUIRE NONE`	TLS is not required for this account, but can still be used.
`REQUIRE SSL`	The account must use TLS, but no valid X509 certificate is required. This option cannot be combined with other TLS options.
`REQUIRE X509`	The account must use TLS and must have a valid X509 certificate. This option implies `REQUIRE SSL`. This option cannot be combined with other TLS options.
`REQUIRE ISSUER 'issuer'`	The account must use TLS and must have a valid X509 certificate. Also, the Certificate Authority must be the one specified via the string `issuer`. This option implies `REQUIRE X509`. This option can be combined with the `SUBJECT`, and `CIPHER` options in any order.
`REQUIRE SUBJECT 'subject'`	The account must use TLS and must have a valid X509 certificate. Also, the certificate's Subject must be the one specified via the string `subject`. This option implies `REQUIRE X509`. This option can be combined with the `ISSUER`, and `CIPHER` options in any order.
`REQUIRE CIPHER 'cipher'`	The account must use TLS, but no valid X509 certificate is required. Also, the encryption used for the connection must use a specific cipher method specified in the string `cipher`. This option implies `REQUIRE SSL`. This option can be combined with the `ISSUER`, and `SUBJECT` options in any order.

The `REQUIRE` keyword must be used only once for all specified options, and the `AND` keyword can be used to separate individual options, but it is not required.

For example, you can create a user account that requires these TLS options with the following:

```
CREATE USER 'alice'@'%'
    REQUIRE SUBJECT '/CN=alice/O=My Dom, Inc./C=US/ST=Oregon/L=Portland'
    AND ISSUER '/C=FI/ST=Somewhere/L=City/ O=Some Company/CN=Peter Parker/emailAddress=p.parker@marvel.com'
    AND CIPHER 'SHA-DES-CBC3-EDH-RSA';
```

If any of these options are set for a specific user account, then any client who tries to connect with that user account will have to be configured to connect with TLS.

See Securing Connections for Client and Server for information on how to enable TLS on the client and server.

Resource Limit Options

MariaDB starting with 10.2.0

MariaDB 10.2.0 introduced a number of resource limit options.

It is possible to set per-account limits for certain server resources. The following table shows the values that can be set per account:

Limit Type	Decription
`MAX_QUERIES_PER_HOUR`	Number of statements that the account can issue per hour (including updates)
`MAX_UPDATES_PER_HOUR`	Number of updates (not queries) that the account can issue per hour
`MAX_CONNECTIONS_PER_HOUR`	Number of connections that the account can start per hour
`MAX_USER_CONNECTIONS`	Number of simultaneous connections that can be accepted from the same account; if it is 0, `max_connections` will be used instead; if `max_connections` is 0, there is no limit for this account's simultaneous connections.
`MAX_STATEMENT_TIME`	Timeout, in seconds, for statements executed by the user. See also Aborting Statements that Exceed a Certain Time to Execute.

If any of these limits are set to `0`, then there is no limit for that resource for that user.

Here is an example showing how to create a user with resource limits:

```
CREATE USER 'someone'@'localhost' WITH
    MAX_USER_CONNECTIONS 10
    MAX_QUERIES_PER_HOUR 200;
```

The resources are tracked per account, which means `'user'@'server'`; not per user name or per connection.

The count can be reset for all users using FLUSH USER_RESOURCES, FLUSH PRIVILEGES or mysqladmin reload.

Per account resource limits are stored in the user table, in the mysql database. Columns used for resources limits are named `max_questions`, `max_updates`, `max_connections` (for `MAX_CONNECTIONS_PER_HOUR`), and `max_user_connections` (for `MAX_USER_CONNECTIONS`).

Account Names

Account names have both a user name component and a host name component, and are specified as `'user_name'@'host_name'`.

The user name and host name may be unquoted, quoted as strings using double quotes (`"`) or single quotes (`'`), or quoted as identifiers using backticks (`` ` ``). You must use quotes when using special characters (such as a hyphen) or wildcard characters. If you quote, you must quote the user name and host name separately (for example `'user_name'@'host_name'`).

Host Name Component

If the host name is not provided, it is assumed to be `'%'`.

Host names may contain the wildcard characters `%` and `_`. They are matched as if by the LIKE clause. If you need to use a wildcard character literally (for example, to match a domain name with an underscore), prefix the character with a backslash. See `LIKE` for more information on escaping wildcard characters.

Host name matches are case-insensitive. Host names can match either domain names or IP addresses. Use `'localhost'` as the host name to allow only local client connections.

You can use a netmask to match a range of IP addresses using `'base_ip/netmask'` as the host name. A user with an IP address *ip_addr* will

be allowed to connect if the following condition is true:

```
ip_addr & netmask = base_ip
```

For example, given a user:

```
CREATE USER 'maria'@'247.150.130.0/255.255.255.0';
```

the IP addresses satisfying this condition range from 247.150.130.0 to 247.150.130.255.

Using `255.255.255.255` is equivalent to not using a netmask at all. Netmasks cannot be used for IPv6 addresses.

Note that the credentials added when creating a user with the `'%'` wildcard host will not grant access in all cases. For example, some systems come with an anonymous localhost user, and when connecting from localhost this will take precedence.

Before MariaDB 10.6, the host name component could be up to 60 characters in length. Starting from MariaDB 10.6, it can be up to 255 characters.

User Name Component

User names must match exactly, including case. A user name that is empty is known as an anonymous account and is allowed to match a login attempt with any user name component. These are described more in the next section.

For valid identifiers to use as user names, see Identifier Names.

It is possible for more than one account to match when a user connects. MariaDB selects the first matching account after sorting according to the following criteria:

- Accounts with an exact host name are sorted before accounts using a wildcard in the host name. Host names using a netmask are considered to be exact for sorting.
- Accounts with a wildcard in the host name are sorted according to the position of the first wildcard character. Those with a wildcard character later in the host name sort before those with a wildcard character earlier in the host name.
- Accounts with a non-empty user name sort before accounts with an empty user name.
- Accounts with an empty user name are sorted last. As mentioned previously, these are known as anonymous accounts. These are described more in the next section.

The following table shows a list of example account as sorted by these criteria:

```
+---------+-------------+
| User    | Host        |
+---------+-------------+
| joffrey | 192.168.0.3 |
|         | 192.168.0.% |
| joffrey | 192.168.%   |
|         | 192.168.%   |
+---------+-------------+
```

Once connected, you only have the privileges granted to the account that matched, not all accounts that could have matched. For example, consider the following commands:

```
CREATE USER 'joffrey'@'192.168.0.3';
CREATE USER 'joffrey'@'%';
GRANT SELECT ON test.t1 to 'joffrey'@'192.168.0.3';
GRANT SELECT ON test.t2 to 'joffrey'@'%';
```

If you connect as joffrey from `192.168.0.3` , you will have the `SELECT` privilege on the table `test.t1` , but not on the table `test.t2` . If you connect as joffrey from any other IP address, you will have the `SELECT` privilege on the table `test.t2` , but not on the table `test.t1` .

Usernames can be up to 80 characters long before 10.6 and starting from 10.6 it can be 128 characters long.

Anonymous Accounts

Anonymous accounts are accounts where the user name portion of the account name is empty. These accounts act as special catch-all accounts. If a user attempts to log into the system from a host, and an anonymous account exists with a host name portion that matches the user's host, then the user will log in as the anonymous account if there is no more specific account match for the user name that the user entered.

For example, here are some anonymous accounts:

```
CREATE USER ''@'localhost';
CREATE USER ''@'192.168.0.3';
```

Fixing a Legacy Default Anonymous Account

On some systems, the mysql.db table has some entries for the `''@'%'` anonymous account by default. Unfortunately, there is no matching entry in the mysql.user/mysql.global_priv_table table, which means that this anonymous account doesn't exactly exist, but it does have privileges--usually on the default `test` database created by mysql_install_db. These account-less privileges are a legacy that is leftover from a time when MySQL's privilege system was less advanced.

This situation means that you will run into errors if you try to create a `''@'%'` account. For example:

```
CREATE USER ''@'%';
ERROR 1396 (HY000): Operation CREATE USER failed for ''@'%'
```

The fix is to DELETE the row in the mysql.db table and then execute FLUSH PRIVILEGES:

```
DELETE FROM mysql.db WHERE User='' AND Host='%';
FLUSH PRIVILEGES;
```

And then the account can be created:

```
CREATE USER ''@'%';
Query OK, 0 rows affected (0.01 sec)
```

See MDEV-13486 for more information.

Password Expiry

MariaDB starting with 10.4.3

Besides automatic password expiry, as determined by default_password_lifetime, password expiry times can be set on an individual user basis, overriding the global setting, for example:

```
CREATE USER 'monty'@'localhost' PASSWORD EXPIRE INTERVAL 120 DAY;
```

See User Password Expiry for more details.

Account Locking

MariaDB starting with 10.4.2

Account locking permits privileged administrators to lock/unlock user accounts. No new client connections will be permitted if an account is locked (existing connections are not affected). For example:

```
CREATE USER 'marijn'@'localhost' ACCOUNT LOCK;
```

See Account Locking for more details.

From MariaDB 10.4.7 and MariaDB 10.5.8, the *lock_option* and *password_option* clauses can occur in either order.

See Also

- Troubleshooting Connection Issues
- Authentication from MariaDB 10.4
- Identifier Names
- GRANT
- ALTER USER
- DROP USER

- SET PASSWORD
- SHOW CREATE USER
- mysql.user table
- mysql.global_priv_table
- Password Validation Plugins - permits the setting of basic criteria for passwords
- Authentication Plugins - allow various authentication methods to be used, and new ones to be developed.

1.1.1.2 ALTER USER

MariaDB starting with 10.2.0

The ALTER USER statement was introduced in MariaDB 10.2.0.

Syntax

```
ALTER USER [IF EXISTS]
 user_specification [,user_specification] ...
  [REQUIRE {NONE | tls_option [[AND] tls_option] ...}]
  [WITH resource_option [resource_option] ...]
  [lock_option] [password_option]

user_specification:
  username [authentication_option]

authentication_option:
  IDENTIFIED BY 'password'
  | IDENTIFIED BY PASSWORD 'password_hash'
  | IDENTIFIED {VIA|WITH} authentication_rule [OR authentication_rule] ...

authentication_rule:
  authentication_plugin
  | authentication_plugin {USING|AS} 'authentication_string'
  | authentication_plugin {USING|AS} PASSWORD('password')

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
  | MAX_STATEMENT_TIME time

password_option:
  PASSWORD EXPIRE
  | PASSWORD EXPIRE DEFAULT
  | PASSWORD EXPIRE NEVER
  | PASSWORD EXPIRE INTERVAL N DAY

lock_option:
    ACCOUNT LOCK
  | ACCOUNT UNLOCK
}
```

Contents

Description

The `ALTER USER` statement modifies existing MariaDB accounts. To use it, you must have the global CREATE USER privilege or the UPDATE privilege for the mysql database. The global SUPER privilege is also required if the read_only system variable is enabled.

If any of the specified user accounts do not yet exist, an error results. If an error occurs, `ALTER USER` will still modify the accounts that do not result in an error. Only one error is produced for all users which have not been modified.

IF EXISTS

When the `IF EXISTS` clause is used, MariaDB will return a warning instead of an error for each specified user that does not exist.

Account Names

For `ALTER USER` statements, account names are specified as the `username` argument in the same way as they are for CREATE USER statements. See account names from the `CREATE USER` page for details on how account names are specified.

CURRENT_USER or `CURRENT_USER()` can also be used to alter the account logged into the current session. For example, to change the current user's password to `mariadb`:

```
ALTER USER CURRENT_USER() IDENTIFIED BY 'mariadb';
```

Authentication Options

MariaDB starting with 10.4

From MariaDB 10.4, it is possible to use more than one authentication plugin for each user account. For example, this can be useful to slowly migrate users to the more secure ed25519 authentication plugin over time, while allowing the old mysql_native_password authentication plugin as an alternative for the transitional period. See Authentication from MariaDB 10.4 for more.

When running `ALTER USER`, not specifying an authentication option in the IDENTIFIED VIA clause will remove that authentication method. (However this was not the case before MariaDB 10.4.13, see MDEV-21928)

For example, a user is created with the ability to authenticate via both a password and unix_socket:

```
CREATE USER 'bob'@'localhost'
  IDENTIFIED VIA mysql_native_password USING PASSWORD('pwd')
  OR unix_socket;

SHOW CREATE USER 'bob'@'localhost'\G
*************************** 1. row ***************************
CREATE USER for bob@localhost: CREATE USER `bob`@`localhost`
  IDENTIFIED VIA mysql_native_password
  USING '*975B2CD4FF9AE554FE8AD33168FBFC326D2021DD'
  OR unix_socket
```

If the user's password is updated, but unix_socket authentication is not specified in the `IDENTIFIED VIA` clause, unix_socket authentication will no longer be permitted.

```
ALTER USER 'bob'@'localhost' IDENTIFIED VIA mysql_native_password
  USING PASSWORD('pwd2');

SHOW CREATE USER 'bob'@'localhost'\G
*************************** 1. row ***************************
CREATE USER for bob@localhost: CREATE USER `bob`@`localhost`
  IDENTIFIED BY PASSWORD '*38366FDA01695B6A5A9DD4E428D9FB8F7EB75512'
```

IDENTIFIED BY 'password'

The optional `IDENTIFIED BY` clause can be used to provide an account with a password. The password should be specified in plain text. It will be hashed by the PASSWORD function prior to being stored to the mysql.user table.

For example, if our password is `mariadb`, then we can set the account's password with:

```
ALTER USER foo2@test IDENTIFIED BY 'mariadb';
```

If you do not specify a password with the `IDENTIFIED BY` clause, the user will be able to connect without a password. A blank password is not a wildcard to match any password. The user must connect without providing a password if no password is set.

The only authentication plugins that this clause supports are mysql_native_password and mysql_old_password.

IDENTIFIED BY PASSWORD 'password_hash'

The optional `IDENTIFIED BY PASSWORD` clause can be used to provide an account with a password that has already been hashed. The password should be specified as a hash that was provided by the PASSWORD#function. It will be stored to the mysql.user table as-is.

For example, if our password is `mariadb`, then we can find the hash with:

```
SELECT PASSWORD('mariadb');
+-------------------------------------------+
| PASSWORD('mariadb')                       |
+-------------------------------------------+
| *54958E764CE10E50764C2EECBB71D01F08549980 |
+-------------------------------------------+
```

And then we can set an account's password with the hash:

```
ALTER USER foo2@test
  IDENTIFIED BY PASSWORD '*54958E764CE10E50764C2EECBB71D01F08549980';
```

If you do not specify a password with the `IDENTIFIED BY` clause, the user will be able to connect without a password. A blank password is not a wildcard to match any password. The user must connect without providing a password if no password is set.

The only authentication plugins that this clause supports are mysql_native_password and mysql_old_password.

IDENTIFIED {VIA|WITH} authentication_plugin

The optional `IDENTIFIED VIA authentication_plugin` allows you to specify that the account should be authenticated by a specific authentication plugin. The plugin name must be an active authentication plugin as per SHOW PLUGINS. If it doesn't show up in that output, then you will need to install it with INSTALL PLUGIN or INSTALL SONAME.

For example, this could be used with the PAM authentication plugin:

```
ALTER USER foo2@test IDENTIFIED VIA pam;
```

Some authentication plugins allow additional arguments to be specified after a `USING` or `AS` keyword. For example, the PAM authentication plugin accepts a service name:

```
ALTER USER foo2@test IDENTIFIED VIA pam USING 'mariadb';
```

The exact meaning of the additional argument would depend on the specific authentication plugin.

In MariaDB 10.4 and later, the `USING` or `AS` keyword can also be used to provide a plain-text password to a plugin if it's provided as an argument to the PASSWORD() function. This is only valid for authentication plugins that have implemented a hook for the PASSWORD() function. For example, the ed25519 authentication plugin supports this:

```
ALTER USER safe@'%' IDENTIFIED VIA ed25519 USING PASSWORD('secret');
```

TLS Options

By default, MariaDB transmits data between the server and clients without encrypting it. This is generally acceptable when the server and client run on the same host or in networks where security is guaranteed through other means. However, in cases where the server and client exist on separate networks or they are in a high-risk network, the lack of encryption does introduce security concerns as a malicious actor could potentially eavesdrop on the traffic as it is sent over the network between them.

To mitigate this concern, MariaDB allows you to encrypt data in transit between the server and clients using the Transport Layer Security (TLS) protocol. TLS was formerly known as Secure Socket Layer (SSL), but strictly speaking the SSL protocol is a predecessor to TLS and, that version of the protocol is now considered insecure. The documentation still uses the term SSL often and for compatibility reasons TLS-related server system and status variables still use the prefix ssl_, but internally, MariaDB only supports its secure successors.

See Secure Connections Overview for more information about how to determine whether your MariaDB server has TLS support.

You can set certain TLS-related restrictions for specific user accounts. For instance, you might use this with user accounts that require access to sensitive data while sending it across networks that you do not control. These restrictions can be enabled for a user account with the CREATE USER, ALTER USER, or GRANT statements. The following options are available:

Option	Description
`REQUIRE NONE`	TLS is not required for this account, but can still be used.
`REQUIRE SSL`	The account must use TLS, but no valid X509 certificate is required. This option cannot be combined with other TLS options.
`REQUIRE X509`	The account must use TLS and must have a valid X509 certificate. This option implies `REQUIRE SSL`. This option cannot be combined with other TLS options.
`REQUIRE ISSUER 'issuer'`	The account must use TLS and must have a valid X509 certificate. Also, the Certificate Authority must be the one specified via the string `issuer`. This option implies `REQUIRE X509`. This option can be combined with the `SUBJECT`, and `CIPHER` options in any order.
`REQUIRE SUBJECT 'subject'`	The account must use TLS and must have a valid X509 certificate. Also, the certificate's Subject must be the one specified via the string `subject`. This option implies `REQUIRE X509`. This option can be combined with the `ISSUER`, and `CIPHER` options in any order.
`REQUIRE CIPHER 'cipher'`	The account must use TLS, but no valid X509 certificate is required. Also, the encryption used for the connection must use a specific cipher method specified in the string `cipher`. This option implies `REQUIRE SSL`. This option can be combined with the `ISSUER`, and `SUBJECT` options in any order.

The `REQUIRE` keyword must be used only once for all specified options, and the `AND` keyword can be used to separate individual options, but it is not required.

For example, you can alter a user account to require these TLS options with the following:

```
ALTER USER 'alice'@'%'
    REQUIRE SUBJECT '/CN=alice/O=My Dom, Inc./C=US/ST=Oregon/L=Portland'
    AND ISSUER '/C=FI/ST=Somewhere/L=City/ O=Some Company/CN=Peter Parker/emailAddress=p.parker@marvel.com'
    AND CIPHER 'SHA-DES-CBC3-EDH-RSA';
```

If any of these options are set for a specific user account, then any client who tries to connect with that user account will have to be configured to connect with TLS.

See Securing Connections for Client and Server for information on how to enable TLS on the client and server.

Resource Limit Options

MariaDB starting with 10.2.0

MariaDB 10.2.0 introduced a number of resource limit options.

It is possible to set per-account limits for certain server resources. The following table shows the values that can be set per account:

Limit Type	Decription
`MAX_QUERIES_PER_HOUR`	Number of statements that the account can issue per hour (including updates)
`MAX_UPDATES_PER_HOUR`	Number of updates (not queries) that the account can issue per hour
`MAX_CONNECTIONS_PER_HOUR`	Number of connections that the account can start per hour
`MAX_USER_CONNECTIONS`	Number of simultaneous connections that can be accepted from the same account; if it is 0, `max_connections` will be used instead; if `max_connections` is 0, there is no limit for this account's simultaneous connections.
`MAX_STATEMENT_TIME`	Timeout, in seconds, for statements executed by the user. See also Aborting Statements that Exceed a Certain Time to Execute.

If any of these limits are set to `0`, then there is no limit for that resource for that user.

Here is an example showing how to set an account's resource limits:

```
ALTER USER 'someone'@'localhost' WITH
    MAX_USER_CONNECTIONS 10
    MAX_QUERIES_PER_HOUR 200;
```

The resources are tracked per account, which means `'user'@'server'`; not per user name or per connection.

The count can be reset for all users using FLUSH USER_RESOURCES, FLUSH PRIVILEGES or mysqladmin reload.

Per account resource limits are stored in the user table, in the mysql database. Columns used for resources limits are named `max_questions`, `max_updates`, `max_connections` (for `MAX_CONNECTIONS_PER_HOUR`), and `max_user_connections` (for `MAX_USER_CONNECTIONS`).

Password Expiry

MariaDB starting with 10.4.3

Besides automatic password expiry, as determined by default_password_lifetime, password expiry times can be set on an individual user basis, overriding the global setting, for example:

```
ALTER USER 'monty'@'localhost' PASSWORD EXPIRE INTERVAL 120 DAY;
ALTER USER 'monty'@'localhost' PASSWORD EXPIRE NEVER;
ALTER USER 'monty'@'localhost' PASSWORD EXPIRE DEFAULT;
```

See User Password Expiry for more details.

Account Locking

MariaDB starting with 10.4.2

Account locking permits privileged administrators to lock/unlock user accounts. No new client connections will be permitted if an account is locked (existing connections are not affected). For example:

```
ALTER USER 'marijn'@'localhost' ACCOUNT LOCK;
```

See Account Locking for more details.

From MariaDB 10.4.7 and MariaDB 10.5.8, the *lock_option* and *password_option* clauses can occur in either order.

See Also

- Authentication from MariaDB 10.4
- GRANT
- CREATE USER
- DROP USER
- SET PASSWORD
- SHOW CREATE USER
- mysql.user table
- Password Validation Plugins - permits the setting of basic criteria for passwords
- Authentication Plugins - allow various authentication methods to be used, and new ones to be developed.

1.1.1.3 DROP USER

Syntax

```
DROP USER [IF EXISTS] user_name [, user_name] ...
```

Contents

Description

The `DROP USER` statement removes one or more MariaDB accounts. It removes privilege rows for the account from all grant tables. To use this statement, you must have the global CREATE USER privilege or the DELETE privilege for the mysql database. Each account is named using the same format as for the `CREATE USER` statement; for example, `'jeffrey'@'localhost'` . If you specify only the user name part of the account name, a host name part of `'%'` is used. For additional information about specifying account names, see CREATE USER.

Note that, if you specify an account that is currently connected, it will not be deleted until the connection is closed. The connection will not be automatically closed.

If any of the specified user accounts do not exist, `ERROR 1396 (HY000)` results. If an error occurs, `DROP USER` will still drop the accounts that do not result in an error. Only one error is produced for all users which have not been dropped:

```
ERROR 1396 (HY000): Operation DROP USER failed for 'u1'@'%','u2'@'%'
```

Failed `CREATE` or `DROP` operations, for both users and roles, produce the same error code.

IF EXISTS

If the `IF EXISTS` clause is used, MariaDB will return a note instead of an error if the user does not exist.

Examples

```
DROP USER bob;
```

`IF EXISTS`:

```
DROP USER bob;
ERROR 1396 (HY000): Operation DROP USER failed for 'bob'@'%'

DROP USER IF EXISTS bob;
Query OK, 0 rows affected, 1 warning (0.00 sec)

SHOW WARNINGS;
+-------+------+-----------------------------------------+
| Level | Code | Message                                 |
+-------+------+-----------------------------------------+
| Note  | 1974 | Can't drop user 'bob'@'%'; it doesn't exist |
+-------+------+-----------------------------------------+
```

See Also

- CREATE USER
- ALTER USER
- GRANT
- SHOW CREATE USER
- mysql.user table

1.1.1.4 GRANT

Contents

Syntax

```
GRANT
    priv_type [(column_list)]
      [, priv_type [(column_list)]] ...
    ON [object_type] priv_level
    TO user_specification [ user_options ...]

user_specification:
  username [authentication_option]

authentication_option:
  IDENTIFIED BY 'password'
  | IDENTIFIED BY PASSWORD 'password_hash'
  | IDENTIFIED {VIA|WITH} authentication_rule [OR authentication_rule  ...]

authentication_rule:
    authentication_plugin
  | authentication_plugin {USING|AS} 'authentication_string'
  | authentication_plugin {USING|AS} PASSWORD('password')

GRANT PROXY ON username
    TO user_specification [, user_specification ...]
    [WITH GRANT OPTION]

GRANT rolename TO grantee [, grantee ...]
    [WITH ADMIN OPTION]

grantee:
    rolename
    username [authentication_option]

user_options:
    [REQUIRE {NONE | tls_option [[AND] tls_option] ...}]
    [WITH with_option [with_option] ...]

object_type:
    TABLE
  | FUNCTION
  | PROCEDURE
  | PACKAGE

priv_level:
    *
  | *.*
  | db_name.*
  | db_name.tbl_name
  | tbl_name
  | db_name.routine_name

with_option:
    GRANT OPTION
  | resource_option

resource_option:
  MAX_QUERIES_PER_HOUR count
  | MAX_UPDATES_PER_HOUR count
  | MAX_CONNECTIONS_PER_HOUR count
  | MAX_USER_CONNECTIONS count
  | MAX_STATEMENT_TIME time

tls_option:
  SSL
  | X509
  | CIPHER 'cipher'
  | ISSUER 'issuer'
  | SUBJECT 'subject'
```

Description

The `GRANT` statement allows you to grant privileges or roles to accounts. To use `GRANT`, you must have the `GRANT OPTION` privilege, and you must have the privileges that you are granting.

Use the REVOKE statement to revoke privileges granted with the `GRANT` statement.

Use the SHOW GRANTS statement to determine what privileges an account has.

Account Names

For `GRANT` statements, account names are specified as the `username` argument in the same way as they are for CREATE USER statements. See account names from the `CREATE USER` page for details on how account names are specified.

Implicit Account Creation

The `GRANT` statement also allows you to implicitly create accounts in some cases.

If the account does not yet exist, then `GRANT` can implicitly create it. To implicitly create an account with `GRANT` , a user is required to have the same privileges that would be required to explicitly create the account with the `CREATE USER` statement.

If the `NO_AUTO_CREATE_USER` SQL_MODE is set, then accounts can only be created if authentication information is specified, or with a CREATE USER statement. If no authentication information is provided, `GRANT` will produce an error when the specified account does not exist, for example:

```
show variables like '%sql_mode%' ;
+---------------+---------------------------------------------+
| Variable_name | Value                                       |
+---------------+---------------------------------------------+
| sql_mode      | NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+---------------+---------------------------------------------+

GRANT USAGE ON *.* TO 'user123'@'%' IDENTIFIED BY '';
ERROR 1133 (28000): Can't find any matching row in the user table

GRANT USAGE ON *.* TO 'user123'@'%' IDENTIFIED VIA PAM using 'mariadb' require ssl ;
Query OK, 0 rows affected (0.00 sec)

select host, user from mysql.user where user='user123' ;

+------+----------+
| host | user     |
+------+----------+
| %    | user123 |
+------+----------+
```

Privilege Levels

Privileges can be set globally, for an entire database, for a table or routine, or for individual columns in a table. Certain privileges can only be set at certain levels.

- Global privileges *priv_type* are granted using `*.*` for *priv_level*. Global privileges include privileges to administer the database and manage user accounts, as well as privileges for all tables, functions, and procedures. Global privileges are stored in the mysql.user table.
- Database privileges *priv_type* are granted using `db_name.*` for *priv_level*, or using just `*` to use default database. Database privileges include privileges to create tables and functions, as well as privileges for all tables, functions, and procedures in the database. Database privileges are stored in the mysql.db table.
- Table privileges *priv_type* are granted using `db_name.tbl_name` for *priv_level*, or using just `tbl_name` to specify a table in the default database. The `TABLE` keyword is optional. Table privileges include the ability to select and change data in the table. Certain table privileges can be granted for individual columns.
- Column privileges *priv_type* are granted by specifying a table for *priv_level* and providing a column list after the privilege type. They allow you to control exactly which columns in a table users can select and change.
- Function privileges *priv_type* are granted using `FUNCTION db_name.routine_name` for *priv_level*, or using just `FUNCTION routine_name` to specify a function in the default database.
- Procedure privileges *priv_type* are granted using `PROCEDURE db_name.routine_name` for *priv_level*, or using just `PROCEDURE routine_name` to specify a procedure in the default database.

The `USAGE` Privilege

The `USAGE` privilege grants no real privileges. The SHOW GRANTS statement will show a global `USAGE` privilege for a newly-created user. You can use `USAGE` with the `GRANT` statement to change options like `GRANT OPTION` and `MAX_USER_CONNECTIONS` without changing any

account privileges.

The `ALL PRIVILEGES` Privilege

The `ALL PRIVILEGES` privilege grants all available privileges. Granting all privileges only affects the given privilege level. For example, granting all privileges on a table does not grant any privileges on the database or globally.

Using `ALL PRIVILEGES` does not grant the special `GRANT OPTION` privilege.

You can use `ALL` instead of `ALL PRIVILEGES`.

The `GRANT OPTION` Privilege

Use the `WITH GRANT OPTION` clause to give users the ability to grant privileges to other users at the given privilege level. Users with the `GRANT OPTION` privilege can only grant privileges they have. They cannot grant privileges at a higher privilege level than they have the `GRANT OPTION` privilege.

The `GRANT OPTION` privilege cannot be set for individual columns. If you use `WITH GRANT OPTION` when specifying column privileges, the `GRANT OPTION` privilege will be granted for the entire table.

Using the `WITH GRANT OPTION` clause is equivalent to listing `GRANT OPTION` as a privilege.

Global Privileges

The following table lists the privileges that can be granted globally. You can also grant all database, table, and function privileges globally. When granted globally, these privileges apply to all databases, tables, or functions, including those created later.

To set a global privilege, use `*.*` for *priv_level*.

BINLOG ADMIN

Enables administration of the binary log, including the PURGE BINARY LOGS statement and setting the binlog_annotate_row_events, binlog_cache_size, binlog_commit_wait_count, binlog_commit_wait_usec, binlog_direct_non_transactional_updates, binlog_expire_logs_seconds, binlog_file_cache_size, binlog_format, binlog_row_image, binlog_row_metadata, binlog_stmt_cache_size, expire_logs_days, log_bin_compress, log_bin_compress_min_len, log_bin_trust_function_creators, max_binlog_cache_size, max_binlog_size, max_binlog_stmt_cache_size, sql_log_bin and sync_binlog system variables. Added in MariaDB 10.5.2.

BINLOG MONITOR

New name for REPLICATION CLIENT from MariaDB 10.5.2, (`REPLICATION CLIENT` still supported as an alias for compatibility purposes). Permits running SHOW commands related to the binary log, in particular the SHOW BINLOG STATUS, SHOW REPLICA STATUS and SHOW BINARY LOGS statements.

BINLOG REPLAY

Enables replaying the binary log with the BINLOG statement (generated by mariadb-binlog), executing SET timestamp when secure_timestamp is set to `replication`, and setting the session values of system variables usually included in BINLOG output, in particular gtid_domain_id, gtid_seq_no, pseudo_thread_id and server_id. Added in MariaDB 10.5.2

CONNECTION ADMIN

Enables administering connection resource limit options. This includes ignoring the limits specified by max_connections, max_user_connections and max_password_errors, not executing the statements specified in init_connect, killing connections and queries owned by other users as well as setting the following connection-related system variables: connect_timeout, disconnect_on_expired_password, extra_max_connections, init_connect, max_connections, max_connect_errors, max_password_errors, proxy_protocol_networks, secure_auth, slow_launch_time, thread_pool_exact_stats, thread_pool_dedicated_listener, thread_pool_idle_timeout, thread_pool_max_threads, thread_pool_min_threads, thread_pool_mode, thread_pool_oversubscribe, thread_pool_prio_kickup_timer, thread_pool_priority, thread_pool_size, thread_pool_stall_limit. Added in MariaDB 10.5.2.

CREATE USER

Create a user using the CREATE USER statement, or implicitly create a user with the `GRANT` statement.

FEDERATED ADMIN

Execute CREATE SERVER, ALTER SERVER, and DROP SERVER statements. Added in MariaDB 10.5.2.

FILE

Read and write files on the server, using statements like LOAD DATA INFILE or functions like LOAD_FILE(). Also needed to create CONNECT outward tables. MariaDB server must have the permissions to access those files.

GRANT OPTION

Grant global privileges. You can only grant privileges that you have.

PROCESS

Show information about the active processes, for example via SHOW PROCESSLIST or mysqladmin processlist. If you have the PROCESS privilege, you can see all threads. Otherwise, you can see only your own threads (that is, threads associated with the MariaDB account that you are using).

READ_ONLY ADMIN

User can set the read_only system variable and allows the user to perform write operations, even when the `read_only` option is active. Added in MariaDB 10.5.2.

RELOAD

Execute FLUSH statements or equivalent mariadb-admin/mysqladmin commands.

REPLICATION CLIENT

Execute SHOW MASTER STATUS, SHOW SLAVE STATUS and SHOW BINARY LOGS informative statements. Renamed to BINLOG MONITOR in MariaDB 10.5.2 (but still supported as an alias for compatibility reasons).

REPLICATION MASTER ADMIN

Permits administration of primary servers, including the SHOW REPLICA HOSTS statement, and setting the gtid_binlog_state, gtid_domain_id, master_verify_checksum and server_id system variables. Added in MariaDB 10.5.2.

REPLICA MONITOR

Permit SHOW REPLICA STATUS and SHOW RELAYLOG EVENTS. From MariaDB 10.5.9.

When a user would upgrade from an older major release to a MariaDB 10.5 minor release prior to MariaDB 10.5.9, certain user accounts would lose capabilities. For example, a user account that had the REPLICATION CLIENT privilege in older major releases could run SHOW REPLICA STATUS, but after upgrading to a MariaDB 10.5 minor release prior to MariaDB 10.5.9, they could no longer run SHOW REPLICA STATUS, because that statement was changed to require the REPLICATION REPLICA ADMIN privilege.

This issue is fixed in MariaDB 10.5.9 with this new privilege, which now grants the user the ability to execute SHOW [ALL] (SLAVE | REPLICA) STATUS.

When a database is upgraded from an older major release to MariaDB Server 10.5.9 or later, any user accounts with the REPLICATION CLIENT or REPLICATION SLAVE privileges will automatically be granted the new REPLICA MONITOR privilege. The privilege fix occurs when the server is started up, not when mariadb-upgrade is performed.

However, when a database is upgraded from an early 10.5 minor release to 10.5.9 and later, the user will have to fix any user account privileges manually.

REPLICATION REPLICA

Synonym for REPLICATION SLAVE. From MariaDB 10.5.1.

REPLICATION SLAVE

Accounts used by replica servers on the primary need this privilege. This is needed to get the updates made on the master. From MariaDB 10.5.1, REPLICATION REPLICA is an alias for `REPLICATION SLAVE`.

REPLICATION SLAVE ADMIN

Permits administering replica servers, including START REPLICA/SLAVE, STOP REPLICA/SLAVE, CHANGE MASTER, SHOW REPLICA/SLAVE STATUS, SHOW RELAYLOG EVENTS statements, replaying the binary log with the BINLOG statement (generated by mariadb-binlog), and setting the gtid_cleanup_batch_size, gtid_ignore_duplicates, gtid_pos_auto_engines, gtid_slave_pos, gtid_strict_mode, init_slave, read_binlog_speed_limit, relay_log_purge, relay_log_recovery, replicate_do_db, replicate_do_table,

replicate_events_marked_for_skip, replicate_ignore_db, replicate_ignore_table, replicate_wild_do_table, replicate_wild_ignore_table, slave_compressed_protocol, slave_ddl_exec_mode, slave_domain_parallel_threads, slave_exec_mode, slave_max_allowed_packet, slave_net_timeout, slave_parallel_max_queued, slave_parallel_mode, slave_parallel_threads, slave_parallel_workers, slave_run_triggers_for_rbr, slave_sql_verify_checksum, slave_transaction_retry_interval, slave_type_conversions, sync_master_info, sync_relay_log and sync_relay_log_info system variables. Added in MariaDB 10.5.2.

SET USER

Enables setting the `DEFINER` when creating triggers, views, stored functions and stored procedures. Added in MariaDB 10.5.2.

SHOW DATABASES

List all databases using the SHOW DATABASES statement. Without the `SHOW DATABASES` privilege, you can still issue the `SHOW DATABASES` statement, but it will only list databases containing tables on which you have privileges.

SHUTDOWN

Shut down the server using SHUTDOWN or the mysqladmin shutdown command.

SUPER

Execute superuser statements: CHANGE MASTER TO, KILL (users who do not have this privilege can only `KILL` their own threads), PURGE LOGS, SET global system variables, or the mysqladmin debug command. Also, this permission allows the user to write data even if the read_only startup option is set, enable or disable logging, enable or disable replication on replica, specify a `DEFINER` for statements that support that clause, connect once after reaching the `MAX_CONNECTIONS` . If a statement has been specified for the init-connect `mysqld` option, that command will not be executed when a user with `SUPER` privileges connects to the server.

The SUPER privilege has been split into multiple smaller privileges from MariaDB 10.5.2 to allow for more fine-grained privileges, although it remains an alias for these smaller privileges.

Database Privileges

The following table lists the privileges that can be granted at the database level. You can also grant all table and function privileges at the database level. Table and function privileges on a database apply to all tables or functions in that database, including those created later.

To set a privilege for a database, specify the database using `db_name.*` for *priv_level*, or just use `*` to specify the default database.

Privilege	Description
`CREATE`	Create a database using the CREATE DATABASE statement, when the privilege is granted for a database. You can grant the `CREATE` privilege on databases that do not yet exist. This also grants the `CREATE` privilege on all tables in the database.
`CREATE ROUTINE`	Create Stored Programs using the CREATE PROCEDURE and CREATE FUNCTION statements.
`CREATE TEMPORARY TABLES`	Create temporary tables with the CREATE TEMPORARY TABLE statement. This privilege enable writing and dropping those temporary tables
`DROP`	Drop a database using the DROP DATABASE statement, when the privilege is granted for a database. This also grants the `DROP` privilege on all tables in the database.
`EVENT`	Create, drop and alter `EVENT` s.
`GRANT OPTION`	Grant database privileges. You can only grant privileges that you have.
`LOCK TABLES`	Acquire explicit locks using the LOCK TABLES statement; you also need to have the `SELECT` privilege on a table, in order to lock it.

Table Privileges

Privilege	Description
`ALTER`	Change the structure of an existing table using the ALTER TABLE statement.
`CREATE`	Create a table using the CREATE TABLE statement. You can grant the `CREATE` privilege on tables that do not yet exist.
`CREATE VIEW`	Create a view using the CREATE_VIEW statement.

`DELETE`	Remove rows from a table using the DELETE statement.
`DELETE HISTORY`	Remove historical rows from a table using the DELETE HISTORY statement. Displays as `DELETE VERSIONING ROWS` when running SHOW GRANTS until MariaDB 10.3.15 and until MariaDB 10.4.5 (MDEV-17655), or when running SHOW PRIVILEGES until MariaDB 10.5.2, MariaDB 10.4.13 and MariaDB 10.3.23 (MDEV-20382). From MariaDB 10.3.4. From MariaDB 10.3.5, if a user has the `SUPER` privilege but not this privilege, running mysql_upgrade will grant this privilege as well.
`DROP`	Drop a table using the DROP TABLE statement or a view using the DROP VIEW statement. Also required to execute the TRUNCATE TABLE statement.
`GRANT OPTION`	Grant table privileges. You can only grant privileges that you have.
`INDEX`	Create an index on a table using the CREATE INDEX statement. Without the `INDEX` privilege, you can still create indexes when creating a table using the CREATE TABLE statement if the you have the `CREATE` privilege, and you can create indexes using the ALTER TABLE statement if you have the `ALTER` privilege.
`INSERT`	Add rows to a table using the INSERT statement. The `INSERT` privilege can also be set on individual columns; see Column Privileges below for details.
`REFERENCES`	Unused.
`SELECT`	Read data from a table using the SELECT statement. The `SELECT` privilege can also be set on individual columns; see Column Privileges below for details.
`SHOW VIEW`	Show the CREATE VIEW statement to create a view using the SHOW CREATE VIEW statement.
`TRIGGER`	Execute triggers associated to tables you update, execute the CREATE TRIGGER and DROP TRIGGER statements. You will still be able to see triggers.
`UPDATE`	Update existing rows in a table using the UPDATE statement. `UPDATE` statements usually include a `WHERE` clause to update only certain rows. You must have `SELECT` privileges on the table or the appropriate columns for the `WHERE` clause. The `UPDATE` privilege can also be set on individual columns; see Column Privileges below for details.

Column Privileges

Some table privileges can be set for individual columns of a table. To use column privileges, specify the table explicitly and provide a list of column names after the privilege type. For example, the following statement would allow the user to read the names and positions of employees, but not other information from the same table, such as salaries.

```
GRANT SELECT (name, position) on Employee to 'jeffrey'@'localhost';
```

Privilege	Description
`INSERT (column_list)`	Add rows specifying values in columns using the INSERT statement. If you only have column-level `INSERT` privileges, you must specify the columns you are setting in the `INSERT` statement. All other columns will be set to their default values, or `NULL`.
`REFERENCES (column_list)`	Unused.
`SELECT (column_list)`	Read values in columns using the SELECT statement. You cannot access or query any columns for which you do not have `SELECT` privileges, including in `WHERE`, `ON`, `GROUP BY`, and `ORDER BY` clauses.
`UPDATE (column_list)`	Update values in columns of existing rows using the UPDATE statement. `UPDATE` statements usually include a `WHERE` clause to update only certain rows. You must have `SELECT` privileges on the table or the appropriate columns for the `WHERE` clause.

Function Privileges

Privilege	Description
`ALTER ROUTINE`	Change the characteristics of a stored function using the ALTER FUNCTION statement.
`EXECUTE`	Use a stored function. You need `SELECT` privileges for any tables or columns accessed by the function.
`GRANT OPTION`	Grant function privileges. You can only grant privileges that you have.

Procedure Privileges

Privilege	Description
`ALTER ROUTINE`	Change the characteristics of a stored procedure using the ALTER PROCEDURE statement.
`EXECUTE`	Execute a stored procedure using the CALL statement. The privilege to call a procedure may allow you to perform actions you wouldn't otherwise be able to do, such as insert rows into a table.
`GRANT OPTION`	Grant procedure privileges. You can only grant privileges that you have.

Proxy Privileges

Privilege	Description
`PROXY`	Permits one user to be a proxy for another.

The `PROXY` privilege allows one user to proxy as another user, which means their privileges change to that of the proxy user, and the CURRENT_USER() function returns the user name of the proxy user.

The `PROXY` privilege only works with authentication plugins that support it. The default mysql_native_password authentication plugin does not support proxy users.

The pam authentication plugin is the only plugin included with MariaDB that currently supports proxy users. The `PROXY` privilege is commonly used with the pam authentication plugin to enable user and group mapping with PAM.

For example, to grant the `PROXY` privilege to an anonymous account that authenticates with the pam authentication plugin, you could execute the following:

```
CREATE USER 'dba'@'%' IDENTIFIED BY 'strongpassword';
GRANT ALL PRIVILEGES ON *.* TO 'dba'@'%' ;

CREATE USER ''@'%' IDENTIFIED VIA pam USING 'mariadb';
GRANT PROXY ON 'dba'@'%' TO ''@'%';
```

A user account can only grant the `PROXY` privilege for a specific user account if the granter also has the `PROXY` privilege for that specific user account, and if that privilege is defined `WITH GRANT OPTION`. For example, the following example fails because the granter does not have the `PROXY` privilege for that specific user account at all:

```
SELECT USER(), CURRENT_USER();
+-----------------+-----------------+
| USER()          | CURRENT_USER()  |
+-----------------+-----------------+
| alice@localhost | alice@localhost |
+-----------------+-----------------+

SHOW GRANTS;
+-------------------------------------------------------------------------------------------------------------------------------+
| Grants for alice@localhost                                                                                                    |
+-------------------------------------------------------------------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'alice'@'localhost' IDENTIFIED BY PASSWORD '*2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19' |
+-------------------------------------------------------------------------------------------------------------------------------+

GRANT PROXY ON 'dba'@'localhost' TO 'bob'@'localhost';
ERROR 1698 (28000): Access denied for user 'alice'@'localhost'
```

And the following example fails because the granter does have the `PROXY` privilege for that specific user account, but it is not defined `WITH GRANT OPTION`:

```
SELECT USER(), CURRENT_USER();
+----------------+----------------+
| USER()         | CURRENT_USER() |
+----------------+----------------+
| alice@localhost | alice@localhost |
+----------------+----------------+

SHOW GRANTS;
+-----------------------------------------------------------------------------------------------------------------------+
| Grants for alice@localhost                                                                                            |
+-----------------------------------------------------------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'alice'@'localhost' IDENTIFIED BY PASSWORD '*2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19' |
| GRANT PROXY ON 'dba'@'localhost' TO 'alice'@'localhost'                                                               |
+-----------------------------------------------------------------------------------------------------------------------+

GRANT PROXY ON 'dba'@'localhost' TO 'bob'@'localhost';
ERROR 1698 (28000): Access denied for user 'alice'@'localhost'
```

But the following example succeeds because the granter does have the `PROXY` privilege for that specific user account, and it is defined `WITH GRANT OPTION`:

```
SELECT USER(), CURRENT_USER();
+----------------+----------------+
| USER()         | CURRENT_USER() |
+----------------+----------------+
| alice@localhost | alice@localhost |
+----------------+----------------+

SHOW GRANTS;
+-----------------------------------------------------------------------------------------------------------------------+
| Grants for alice@localhost                                                                                            |
+-----------------------------------------------------------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'alice'@'localhost' IDENTIFIED BY PASSWORD '*2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19' WITH |
| GRANT PROXY ON 'dba'@'localhost' TO 'alice'@'localhost' WITH GRANT OPTION                                             |
+-----------------------------------------------------------------------------------------------------------------------+

GRANT PROXY ON 'dba'@'localhost' TO 'bob'@'localhost';
```

A user account can grant the `PROXY` privilege for any other user account if the granter has the `PROXY` privilege for the `''@'%'` anonymous user account, like this:

```
GRANT PROXY ON ''@'%' TO 'dba'@'localhost' WITH GRANT OPTION;
```

For example, the following example succeeds because the user can grant the `PROXY` privilege for any other user account:

```
SELECT USER(), CURRENT_USER();
+----------------+----------------+
| USER()         | CURRENT_USER() |
+----------------+----------------+
| alice@localhost | alice@localhost |
+----------------+----------------+

SHOW GRANTS;
+-----------------------------------------------------------------------------------------------------------------------+
| Grants for alice@localhost                                                                                            |
+-----------------------------------------------------------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'alice'@'localhost' IDENTIFIED BY PASSWORD '*2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19' WITH |
| GRANT PROXY ON ''@'%' TO 'alice'@'localhost' WITH GRANT OPTION                                                        |
+-----------------------------------------------------------------------------------------------------------------------+

GRANT PROXY ON 'app1_dba'@'localhost' TO 'bob'@'localhost';
Query OK, 0 rows affected (0.004 sec)

GRANT PROXY ON 'app2_dba'@'localhost' TO 'carol'@'localhost';
Query OK, 0 rows affected (0.004 sec)
```

The default `root` user accounts created by mysql_install_db have this privilege. For example:

```
GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION;
GRANT PROXY ON ''@'%' TO 'root'@'localhost' WITH GRANT OPTION;
```

This allows the default `root` user accounts to grant the `PROXY` privilege for any other user account, and it also allows the default `root` user accounts to grant others the privilege to do the same.

Authentication Options

The authentication options for the `GRANT` statement are the same as those for the CREATE USER statement.

IDENTIFIED BY 'password'

The optional `IDENTIFIED BY` clause can be used to provide an account with a password. The password should be specified in plain text. It will be hashed by the PASSWORD function prior to being stored to the mysql.user table.

For example, if our password is `mariadb`, then we can create the user with:

```
GRANT USAGE ON *.* TO foo2@test IDENTIFIED BY 'mariadb';
```

If you do not specify a password with the `IDENTIFIED BY` clause, the user will be able to connect without a password. A blank password is not a wildcard to match any password. The user must connect without providing a password if no password is set.

If the user account already exists and if you provide the `IDENTIFIED BY` clause, then the user's password will be changed. You must have the privileges needed for the SET PASSWORD statement to change a user's password with `GRANT`.

The only authentication plugins that this clause supports are mysql_native_password and mysql_old_password.

IDENTIFIED BY PASSWORD 'password_hash'

The optional `IDENTIFIED BY PASSWORD` clause can be used to provide an account with a password that has already been hashed. The password should be specified as a hash that was provided by the PASSWORD function. It will be stored to the mysql.user table as-is.

For example, if our password is `mariadb`, then we can find the hash with:

```
SELECT PASSWORD('mariadb');
+-------------------------------------------+
| PASSWORD('mariadb')                       |
+-------------------------------------------+
| *54958E764CE10E50764C2EECBB71D01F08549980 |
+-------------------------------------------+
1 row in set (0.00 sec)
```

And then we can create a user with the hash:

```
GRANT USAGE ON *.* TO foo2@test IDENTIFIED BY PASSWORD '*54958E764CE10E50764C2EECBB71D01F08549980';
```

If you do not specify a password with the `IDENTIFIED BY` clause, the user will be able to connect without a password. A blank password is not a wildcard to match any password. The user must connect without providing a password if no password is set.

If the user account already exists and if you provide the `IDENTIFIED BY` clause, then the user's password will be changed. You must have the privileges needed for the SET PASSWORD statement to change a user's password with `GRANT`.

The only authentication plugins that this clause supports are mysql_native_password and mysql_old_password.

IDENTIFIED {VIA|WITH} authentication_plugin

The optional `IDENTIFIED VIA authentication_plugin` allows you to specify that the account should be authenticated by a specific authentication plugin. The plugin name must be an active authentication plugin as per SHOW PLUGINS. If it doesn't show up in that output, then you will need to install it with INSTALL PLUGIN or INSTALL SONAME.

For example, this could be used with the PAM authentication plugin:

```
GRANT USAGE ON *.* TO foo2@test IDENTIFIED VIA pam;
```

Some authentication plugins allow additional arguments to be specified after a `USING` or `AS` keyword. For example, the PAM authentication plugin accepts a service name:

```
GRANT USAGE ON *.* TO foo2@test IDENTIFIED VIA pam USING 'mariadb';
```

The exact meaning of the additional argument would depend on the specific authentication plugin.

MariaDB starting with 10.4.0

The `USING` or `AS` keyword can also be used to provide a plain-text password to a plugin if it's provided as an argument to the PASSWORD() function. This is only valid for authentication plugins that have implemented a hook for the PASSWORD() function. For example, the ed25519 authentication plugin supports this:

```
CREATE USER safe@'%' IDENTIFIED VIA ed25519 USING PASSWORD('secret');
```

MariaDB starting with 10.4.3

One can specify many authentication plugins, they all work as alternatives ways of authenticating a user:

```
CREATE USER safe@'%' IDENTIFIED VIA ed25519 USING PASSWORD('secret') OR unix_socket;
```

By default, when you create a user without specifying an authentication plugin, MariaDB uses the mysql_native_password plugin.

Resource Limit Options

MariaDB starting with 10.2.0

MariaDB 10.2.0 introduced a number of resource limit options.

It is possible to set per-account limits for certain server resources. The following table shows the values that can be set per account:

Limit Type	Decription
`MAX_QUERIES_PER_HOUR`	Number of statements that the account can issue per hour (including updates)
`MAX_UPDATES_PER_HOUR`	Number of updates (not queries) that the account can issue per hour
`MAX_CONNECTIONS_PER_HOUR`	Number of connections that the account can start per hour
`MAX_USER_CONNECTIONS`	Number of simultaneous connections that can be accepted from the same account; if it is 0, `max_connections` will be used instead; if `max_connections` is 0, there is no limit for this account's simultaneous connections.
`MAX_STATEMENT_TIME`	Timeout, in seconds, for statements executed by the user. See also Aborting Statements that Exceed a Certain Time to Execute.

If any of these limits are set to `0`, then there is no limit for that resource for that user.

To set resource limits for an account, if you do not want to change that account's privileges, you can issue a `GRANT` statement with the `USAGE` privilege, which has no meaning. The statement can name some or all limit types, in any order.

Here is an example showing how to set resource limits:

```
GRANT USAGE ON *.* TO 'someone'@'localhost' WITH
    MAX_USER_CONNECTIONS 0
    MAX_QUERIES_PER_HOUR 200;
```

The resources are tracked per account, which means `'user'@'server'`; not per user name or per connection.

The count can be reset for all users using FLUSH USER_RESOURCES, FLUSH PRIVILEGES or mysqladmin reload.

Users with the `CONNECTION ADMIN` privilege (in MariaDB 10.5.2 and later) or the `SUPER` privilege are not restricted by `max_user_connections`, `max_connections`, or `max_password_errors`.

Per account resource limits are stored in the user table, in the mysql database. Columns used for resources limits are named `max_questions`, `max_updates`, `max_connections` (for `MAX_CONNECTIONS_PER_HOUR`), and `max_user_connections` (for `MAX_USER_CONNECTIONS`).

TLS Options

By default, MariaDB transmits data between the server and clients without encrypting it. This is generally acceptable when the server and client run on the same host or in networks where security is guaranteed through other means. However, in cases where the server and client exist on separate networks or they are in a high-risk network, the lack of encryption does introduce security concerns as a malicious actor could potentially eavesdrop on the traffic as it is sent over the network between them.

To mitigate this concern, MariaDB allows you to encrypt data in transit between the server and clients using the Transport Layer Security (TLS) protocol. TLS was formerly known as Secure Socket Layer (SSL), but strictly speaking the SSL protocol is a predecessor to TLS and, that version of the protocol is now considered insecure. The documentation still uses the term SSL often and for compatibility reasons TLS-related server system and status variables still use the prefix ssl_, but internally, MariaDB only supports its secure successors.

See Secure Connections Overview for more information about how to determine whether your MariaDB server has TLS support.

You can set certain TLS-related restrictions for specific user accounts. For instance, you might use this with user accounts that require access to sensitive data while sending it across networks that you do not control. These restrictions can be enabled for a user account with the CREATE USER, ALTER USER, or GRANT statements. The following options are available:

Option	Description
`REQUIRE NONE`	TLS is not required for this account, but can still be used.
`REQUIRE SSL`	The account must use TLS, but no valid X509 certificate is required. This option cannot be combined with other TLS options.
`REQUIRE X509`	The account must use TLS and must have a valid X509 certificate. This option implies `REQUIRE SSL`. This option cannot be combined with other TLS options.
`REQUIRE ISSUER 'issuer'`	The account must use TLS and must have a valid X509 certificate. Also, the Certificate Authority must be the one specified via the string `issuer`. This option implies `REQUIRE X509`. This option can be combined with the `SUBJECT`, and `CIPHER` options in any order.
`REQUIRE SUBJECT 'subject'`	The account must use TLS and must have a valid X509 certificate. Also, the certificate's Subject must be the one specified via the string `subject`. This option implies `REQUIRE X509`. This option can be combined with the `ISSUER`, and `CIPHER` options in any order.
`REQUIRE CIPHER 'cipher'`	The account must use TLS, but no valid X509 certificate is required. Also, the encryption used for the connection must use a specific cipher method specified in the string `cipher`. This option implies `REQUIRE SSL`. This option can be combined with the `ISSUER`, and `SUBJECT` options in any order.

The `REQUIRE` keyword must be used only once for all specified options, and the `AND` keyword can be used to separate individual options, but it is not required.

For example, you can create a user account that requires these TLS options with the following:

```
GRANT USAGE ON *.* TO 'alice'@'%'
    REQUIRE SUBJECT '/CN=alice/O=My Dom, Inc./C=US/ST=Oregon/L=Portland'
    AND ISSUER '/C=FI/ST=Somewhere/L=City/ O=Some Company/CN=Peter Parker/emailAddress=p.parker@marvel.com'
    AND CIPHER 'SHA-DES-CBC3-EDH-RSA';
```

If any of these options are set for a specific user account, then any client who tries to connect with that user account will have to be configured to connect with TLS.

See Securing Connections for Client and Server for information on how to enable TLS on the client and server.

Roles

Syntax

```
GRANT role TO grantee [, grantee ... ]
[ WITH ADMIN OPTION ]

grantee:
    rolename
    username [authentication_option]
```

The GRANT statement is also used to grant the use a role to one or more users or other roles. In order to be able to grant a role, the grantor doing so must have permission to do so (see WITH ADMIN in the CREATE ROLE article).

Specifying the `WITH ADMIN OPTION` permits the grantee to in turn grant the role to another.

For example, the following commands show how to grant the same role to a couple different users.

```
GRANT journalist TO hulda;

GRANT journalist TO berengar WITH ADMIN OPTION;
```

If a user has been granted a role, they do not automatically obtain all permissions associated with that role. These permissions are only in use when the user activates the role with the SET ROLE statement.

Grant Examples

Granting Root-like Privileges

You can create a user that has privileges similar to the default `root` accounts by executing the following:

```
CREATE USER 'alexander'@'localhost';
GRANT ALL PRIVILEGES ON  *.* to 'alexander'@'localhost' WITH GRANT OPTION;
```

See Also

- Troubleshooting Connection Issues
- --skip-grant-tables allows you to start MariaDB without `GRANT` . This is useful if you lost your root password.
- CREATE USER
- ALTER USER
- DROP USER
- SET PASSWORD
- SHOW CREATE USER
- mysql.user table
- Password Validation Plugins - permits the setting of basic criteria for passwords
- Authentication Plugins - allow various authentication methods to be used, and new ones to be developed.

1.1.1.5 RENAME USER

Syntax

```
RENAME USER old_user TO new_user
    [, old_user TO new_user] ...
```

Description

The RENAME USER statement renames existing MariaDB accounts. To use it, you must have the global `CREATE USER` privilege or the `UPDATE` privilege for the `mysql` database. Each account is named using the same format as for the `CREATE USER` statement; for example, `'jeffrey'@'localhost'` . If you specify only the user name part of the account name, a host name part of `'%'` is used.

If any of the old user accounts do not exist or any of the new user accounts already exist, `ERROR 1396 (HY000)` results. If an error occurs, `RENAME USER` will still rename the accounts that do not result in an error.

Examples

```
CREATE USER 'donald', 'mickey';
RENAME USER 'donald' TO 'duck'@'localhost', 'mickey' TO 'mouse'@'localhost';
```

1.1.1.6 REVOKE

Contents

Privileges

Syntax

```
REVOKE
    priv_type [(column_list)]
      [, priv_type [(column_list)]] ...
    ON [object_type] priv_level
    FROM user [, user] ...

REVOKE ALL PRIVILEGES, GRANT OPTION
    FROM user [, user] ...
```

Description

The `REVOKE` statement enables system administrators to revoke privileges (or roles - see section below) from MariaDB accounts. Each account is named using the same format as for the `GRANT` statement; for example, '`jeffrey'@'localhost`'. If you specify only the user name part of the account name, a host name part of '`%`' is used. For details on the levels at which privileges exist, the allowable `priv_type` and `priv_level` values, and the syntax for specifying users and passwords, see GRANT.

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 user or users:

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM user [, user] ...
```

To use this `REVOKE` syntax, you must have the global CREATE USER privilege or the UPDATE privilege for the mysql database. See GRANT.

Examples

```
REVOKE SUPER ON *.* FROM 'alexander'@'localhost';
```

Roles

Syntax

```
REVOKE role  [, role ...]
    FROM grantee [, grantee2 ... ]

REVOKE ADMIN OPTION FOR role FROM grantee [, grantee2]
```

Description

`REVOKE` is also used to remove a role from a user or another role that it's previously been assigned to. If a role has previously been set as a

default role, `REVOKE` does not remove the record of the default role from the mysql.user table. If the role is subsequently granted again, it will again be the user's default. Use SET DEFAULT ROLE NONE to explicitly remove this.

Before MariaDB 10.1.13, the `REVOKE role` statement was not permitted in prepared statements.

Example

```
REVOKE journalist FROM hulda
```

1.1.1.7 SET PASSWORD

Syntax

```
SET PASSWORD [FOR user] =
    {
        PASSWORD('some password')
      | OLD_PASSWORD('some password')
      | 'encrypted password'
    }
```

Contents

Description

The `SET PASSWORD` statement assigns a password to an existing MariaDB user account.

If the password is specified using the `PASSWORD()` or `OLD_PASSWORD()` function, the literal text of the password should be given. If the password is specified without using either function, the password should be the already-encrypted password value as returned by `PASSWORD()`.

`OLD_PASSWORD()` should only be used if your MariaDB/MySQL clients are very old (< 4.0.0).

With no `FOR` clause, this statement sets the password for the current user. Any client that has connected to the server using a non-anonymous account can change the password for that account.

With a `FOR` clause, this statement sets the password for a specific account on the current server host. Only clients that have the `UPDATE` privilege for the `mysql` database can do this. The user value should be given in `user_name@host_name` format, where `user_name` and `host_name` are exactly as they are listed in the User and Host columns of the `mysql.user` table entry.

The argument to `PASSWORD()` and the password given to MariaDB clients can be of arbitrary length.

Authentication Plugin Support

MariaDB starting with 10.4

In MariaDB 10.4 and later, `SET PASSWORD` (with or without `PASSWORD()`) works for accounts authenticated via any authentication plugin that supports passwords stored in the `mysql.global_priv` table.

The `ed25519`, `mysql_native_password`, and `mysql_old_password` authentication plugins store passwords in the `mysql.global_priv` table.

If you run `SET PASSWORD` on an account that authenticates with one of these authentication plugins that stores passwords in the `mysql.global_priv` table, then the `PASSWORD()` function is evaluated by the specific authentication plugin used by the account. The authentication plugin hashes the password with a method that is compatible with that specific authentication plugin.

The `unix_socket`, `named_pipe`, `gssapi`, and `pam` authentication plugins do **not** store passwords in the `mysql.global_priv` table. These

authentication plugins rely on other methods to authenticate the user.

If you attempt to run `SET PASSWORD` on an account that authenticates with one of these authentication plugins that doesn't store a password in the `mysql.global_priv` table, then MariaDB Server will raise a warning like the following:

```
SET PASSWORD is ignored for users authenticating via unix_socket plugin
```

See Authentication from MariaDB 10.4 for an overview of authentication changes in MariaDB 10.4.

MariaDB until 10.3

In MariaDB 10.3 and before, `SET PASSWORD` (with or without `PASSWORD()`) only works for accounts authenticated via `mysql_native_password` or `mysql_old_password` authentication plugins

Passwordless User Accounts

User accounts do not always require passwords to login.

The `unix_socket`, `named_pipe` and `gssapi` authentication plugins do not require a password to authenticate the user.

The `pam` authentication plugin may or may not require a password to authenticate the user, depending on the specific configuration.

The `mysql_native_password` and `mysql_old_password` authentication plugins require passwords for authentication, but the password can be blank. In that case, no password is required.

If you provide a password while attempting to log into the server as an account that doesn't require a password, then MariaDB server will simply ignore the password.

MariaDB starting with 10.4

In MariaDB 10.4 and later, a user account can be defined to use multiple authentication plugins in a specific order of preference. This specific scenario may be more noticeable in these versions, since an account could be associated with some authentication plugins that require a password, and some that do not.

Example

For example, if you had an entry with User and Host column values of '`bob`' and '`%.loc.gov`', you would write the statement like this:

```
SET PASSWORD FOR 'bob'@'%.loc.gov' = PASSWORD('newpass');
```

If you want to delete a password for a user, you would do:

```
SET PASSWORD FOR 'bob'@localhost = PASSWORD("");
```

See Also

- Password Validation Plugins - permits the setting of basic criteria for passwords
- `ALTER USER`

1.1.1.8 CREATE ROLE

Syntax

```
CREATE [OR REPLACE] ROLE [IF NOT EXISTS] role
  [WITH ADMIN
    {CURRENT_USER | CURRENT_ROLE | user | role}]
```

Contents

Description

The `CREATE ROLE` statement creates one or more MariaDB roles. To use it, you must have the global CREATE USER privilege or the INSERT privilege for the mysql database. For each account, `CREATE ROLE` creates a new row in the mysql.user table that has no privileges, and with the corresponding `is_role` field set to `Y`. It also creates a record in the mysql.roles_mapping table.

If any of the specified roles already exist, `ERROR 1396 (HY000)` results. If an error occurs, `CREATE ROLE` will still create the roles that do not result in an error. The maximum length for a role is 128 characters. Role names can be quoted, as explained in the Identifier names page. Only one error is produced for all roles which have not been created:

```
ERROR 1396 (HY000): Operation CREATE ROLE failed for 'a','b','c'
```

Failed `CREATE` or `DROP` operations, for both users and roles, produce the same error code.

`PUBLIC` and `NONE` are reserved, and cannot be used as role names. `NONE` is used to unset a role and `PUBLIC` has a special use in other systems, such as Oracle, so is reserved for compatibility purposes.

Before MariaDB 10.1.13, the `CREATE ROLE` statement was not permitted in prepared statements.

For valid identifiers to use as role names, see Identifier Names.

WITH ADMIN

The optional `WITH ADMIN` clause determines whether the current user, the current role or another user or role has use of the newly created role. If the clause is omitted, `WITH ADMIN CURRENT_USER` is treated as the default, which means that the current user will be able to GRANT this role to users.

OR REPLACE

If the optional `OR REPLACE` clause is used, it acts as a shortcut for:

```
DROP ROLE IF EXISTS name;
CREATE ROLE name ...;
```

IF NOT EXISTS

When the `IF NOT EXISTS` clause is used, MariaDB will return a warning instead of an error if the specified role already exists. Cannot be used together with the `OR REPLACE` clause.

Examples

```
CREATE ROLE journalist;

CREATE ROLE developer WITH ADMIN lorinda@localhost;
```

Granting the role to another user. Only user `lorinda@localhost` has permission to grant the `developer` role:

```
 SELECT USER();
+-------------------+
| USER()            |
+-------------------+
| henning@localhost |
+-------------------+
...
GRANT developer TO ian@localhost;
Access denied for user 'henning'@'localhost'

 SELECT USER();
+-------------------+
| USER()            |
+-------------------+
| lorinda@localhost |
+-------------------+

GRANT m_role TO ian@localhost;
```

The `OR REPLACE` and `IF NOT EXISTS` clauses. The `journalist` role already exists:

```
CREATE ROLE journalist;
ERROR 1396 (HY000): Operation CREATE ROLE failed for 'journalist'

CREATE OR REPLACE ROLE journalist;
Query OK, 0 rows affected (0.00 sec)

CREATE ROLE IF NOT EXISTS journalist;
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

```
SHOW WARNINGS;
+-------+------+-----------------------------------------------------+
| Level | Code | Message                                             |
+-------+------+-----------------------------------------------------+
| Note  | 1975 | Can't create role 'journalist'; it already exists |
+-------+------+-----------------------------------------------------+
```

See Also

- Identifier Names
- Roles Overview
- DROP ROLE

1.1.1.9 DROP ROLE

Syntax

```
DROP ROLE [IF EXISTS] role_name [,role_name ...]
```

Contents

Description

The `DROP ROLE` statement removes one or more MariaDB roles. To use this statement, you must have the global CREATE USER privilege or

the DELETE privilege for the mysql database.

`DROP ROLE` does not disable roles for connections which selected them with SET ROLE. If a role has previously been set as a default role, `DROP ROLE` does not remove the record of the default role from the mysql.user table. If the role is subsequently recreated and granted, it will again be the user's default. Use SET DEFAULT ROLE NONE to explicitly remove this.

If any of the specified user accounts do not exist, `ERROR 1396 (HY000)` results. If an error occurs, `DROP ROLE` will still drop the roles that do not result in an error. Only one error is produced for all roles which have not been dropped:

```
ERROR 1396 (HY000): Operation DROP ROLE failed for 'a','b','c'
```

Failed `CREATE` or `DROP` operations, for both users and roles, produce the same error code.

Before MariaDB 10.1.13, the `DROP ROLE` statement was not permitted in prepared statements.

IF EXISTS

If the `IF EXISTS` clause is used, MariaDB will return a warning instead of an error if the role does not exist.

Examples

```
DROP ROLE journalist;
```

The same thing using the optional `IF EXISTS` clause:

```
DROP ROLE journalist;
ERROR 1396 (HY000): Operation DROP ROLE failed for 'journalist'

DROP ROLE IF EXISTS journalist;
Query OK, 0 rows affected, 1 warning (0.00 sec)

Note (Code 1975): Can't drop role 'journalist'; it doesn't exist
```

See Also

- Roles Overview
- CREATE ROLE

1.1.1.10 SET ROLE

Syntax

```
SET ROLE { role | NONE }
```

Contents

Description

The `SET ROLE` statement enables a role, along with all of its associated permissions, for the current session. To unset a role, use `NONE` .

If a role that doesn't exist, or to which the user has not been assigned, is specified, an `ERROR 1959 (OP000): Invalid role specification` error occurs.

An automatic SET ROLE is implicitly performed when a user connects if that user has been assigned a default role. See SET DEFAULT ROLE.

Example

```
SELECT CURRENT_ROLE;
+--------------+
| CURRENT_ROLE |
+--------------+
| NULL         |
+--------------+

SET ROLE staff;

SELECT CURRENT_ROLE;
+--------------+
| CURRENT_ROLE |
+--------------+
| staff        |
+--------------+

SET ROLE NONE;

SELECT CURRENT_ROLE();
+----------------+
| CURRENT_ROLE() |
+----------------+
| NULL           |
+----------------+
```

1.1.1.11 SET DEFAULT ROLE

Contents

Syntax

```
SET DEFAULT ROLE { role | NONE } [ FOR user@host ]
```

Description

The `SET DEFAULT ROLE` statement sets a **default role** for a specified (or current) user. A default role is automatically enabled when a user connects (an implicit SET ROLE statement is executed immediately after a connection is established).

To be able to set a role as a default, the role must already have been granted to that user, and one needs the privileges to enable this role (if you cannot do `SET ROLE X`, you won't be able to do `SET DEFAULT ROLE X`). To set a default role for another user one needs to have write access to the `mysql` database.

To remove a user's default role, use `SET DEFAULT ROLE NONE [ FOR user@host ]`. The record of the default role is not removed if the role is dropped or revoked, so if the role is subsequently re-created or granted, it will again be the user's default role.

The default role is stored in the `default_role` column in the mysql.user table/view, as well as in the Information Schema APPLICABLE_ROLES table, so these can be viewed to see which role has been assigned to a user as the default.

Examples

Setting a default role for the current user:

```
SET DEFAULT ROLE journalist;
```

Removing a default role from the current user:

```
SET DEFAULT ROLE NONE;
```

Setting a default role for another user. The role has to have been granted to the user before it can be set as default:

```
CREATE ROLE journalist;
CREATE USER taniel;

SET DEFAULT ROLE journalist FOR taniel;
ERROR 1959 (OP000): Invalid role specification `journalist`

GRANT journalist TO taniel;
SET DEFAULT ROLE journalist FOR taniel;
```

Viewing mysql.user:

```
select * from mysql.user where user='taniel'\G
*************************** 1. row ***************************
                 Host: %
                 User: taniel
...
              is_role: N
         default_role: journalist
...
```

Removing a default role for another user

```
SET DEFAULT ROLE NONE FOR taniel;
```

1.1.1.12 SHOW GRANTS

Contents

Syntax

```
SHOW GRANTS [FOR user|role]
```

Description

The `SHOW GRANTS` statement lists privileges granted to a particular user or role.

Users

The statement lists the GRANT statement or statements that must be issued to duplicate the privileges that are granted to a MariaDB user account. The account is named using the same format as for the `GRANT` statement; for example, '`jeffrey'@'localhost`'. If you specify only the user name part of the account name, a host name part of '`%`' is used. For additional information about specifying account names, see GRANT.

```
SHOW GRANTS FOR 'root'@'localhost';
+---------------------------------------------------------------------+
| Grants for root@localhost                                           |
+---------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION |
+---------------------------------------------------------------------+
```

To list the privileges granted to the account that 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 of the equivalent syntaxes) is used in `DEFINER` context (such as within a stored procedure that is defined with `SQL SECURITY DEFINER`), the grants displayed are those of the definer and not the invoker.

Note that the `DELETE HISTORY` privilege, introduced in MariaDB 10.3.4, was displayed as `DELETE VERSIONING ROWS` when running `SHOW GRANTS` until MariaDB 10.3.15 (MDEV-17655).

Roles

`SHOW GRANTS` can also be used to view the privileges granted to a role.

Example

```
SHOW GRANTS FOR journalist;
+----------------------------------------+
| Grants for journalist                  |
+----------------------------------------+
| GRANT USAGE ON *.* TO 'journalist'     |
| GRANT DELETE ON `test`.* TO 'journalist' |
+----------------------------------------+
```

See Also

- Authentication from MariaDB 10.4
- SHOW CREATE USER shows how the user was created.
- SHOW PRIVILEGES shows the privileges supported by MariaDB.
- Roles

1.1.1.13 SHOW CREATE USER

MariaDB starting with 10.2.0

`SHOW CREATE USER` was introduced in MariaDB 10.2.0

Syntax

```
SHOW CREATE USER user_name
```

Description

Shows the CREATE USER statement that created the given user. The statement requires the SELECT privilege for the mysql database, except for the current user.

Examples

```
CREATE USER foo4@test require cipher 'text'
  issuer 'foo_issuer' subject 'foo_subject';

SHOW CREATE USER foo4@test\G
*************************** 1. row ***************************
CREATE USER 'foo4'@'test'
  REQUIRE ISSUER 'foo_issuer'
  SUBJECT 'foo_subject'
  CIPHER 'text'
```

User Password Expiry:

```
CREATE USER 'monty'@'localhost' PASSWORD EXPIRE INTERVAL 120 DAY;

SHOW CREATE USER 'monty'@'localhost';
+-------------------------------------------------------------------+
| CREATE USER for monty@localhost                                   |
+-------------------------------------------------------------------+
| CREATE USER 'monty'@'localhost' PASSWORD EXPIRE INTERVAL 120 DAY |
+-------------------------------------------------------------------+
```

See Also

- CREATE USER
- ALTER USER
- SHOW GRANTS shows the `GRANTS/PRIVILEGES` for a user.
- SHOW PRIVILEGES shows the privileges supported by MariaDB.

1.1.2 Administrative SQL Statements

1.1.2.1 Table Statements

Articles about creating, modifying, and maintaining tables in MariaDB.

ALTER
The various ALTER statements in MariaDB.

ANALYZE TABLE
Store key distributions for a table.

CHECK TABLE
Check table for errors.

CHECK VIEW
Check whether the view algorithm is correct.

CHECKSUM TABLE
Report a table checksum.

CREATE TABLE
Creates a new table.

DELETE
Delete rows from one or more tables.

DROP TABLE
Removes definition and data from one or more tables.

Installing System Tables (mysql_install_db)
Using mysql_install_db to create the system tables in the 'mysql' database directory

mysqlcheck
Tool for checking, repairing, analyzing and optimizing tables.

mysql_upgrade
Update to the latest version.

OPTIMIZE TABLE
Reclaim unused space and defragment data.

RENAME TABLE
Change a table's name.

REPAIR TABLE
Rapairs a table, if the storage engine supports this statement.

REPAIR VIEW
Fix view if the algorithms are swapped.

REPLACE
Equivalent to DELETE + INSERT, or just an INSERT if no rows are returned.

SHOW COLUMNS
Column information.

SHOW CREATE TABLE
Shows the CREATE TABLE statement that created the table.

SHOW INDEX
Information about table indexes.

TRUNCATE TABLE
DROP and re-CREATE a table.

UPDATE
Modify rows in one or more tables.

Obsolete Table Commands
Table commands that have been removed from MariaDB

IGNORE
Suppress errors while trying to violate a UNIQUE constraint.

System-Versioned Tables
System-versioned tables record the history of all changes to table data.

1.1.2.1.1 ALTER

1.1.2.1.1.1 ALTER TABLE

Syntax

```
ALTER [ONLINE] [IGNORE] TABLE [IF EXISTS] tbl_name
    [WAIT n | NOWAIT]
    alter_specification [, alter_specification] ...

alter_specification:
    table_option ...
  | ADD [COLUMN] [IF NOT EXISTS] col_name column_definition
        [FIRST | AFTER col_name ]
  | ADD [COLUMN] [IF NOT EXISTS] (col_name column_definition,...)
  | ADD {INDEX|KEY} [IF NOT EXISTS] [index_name]
        [index_type] (index_col_name,...) [index_option] ...
  | ADD [CONSTRAINT [symbol]] PRIMARY KEY
        [index_type] (index_col_name,...) [index_option] ...
  | ADD [CONSTRAINT [symbol]]
        UNIQUE [INDEX|KEY] [index_name]
        [index_type] (index_col_name,...) [index_option] ...
  | ADD FULLTEXT [INDEX|KEY] [index_name]
        (index_col_name,...) [index_option] ...
  | ADD SPATIAL [INDEX|KEY] [index_name]
        (index_col_name,...) [index_option] ...
  | ADD [CONSTRAINT [symbol]]
        FOREIGN KEY [IF NOT EXISTS] [index_name] (index_col_name,...)
        reference_definition
  | ADD PERIOD FOR SYSTEM_TIME (start_column_name, end_column_name)
  | ALTER [COLUMN] col_name SET DEFAULT literal | (expression)
  | ALTER [COLUMN] col_name DROP DEFAULT
  | ALTER {INDEX|KEY} index_name [NOT] INVISIBLE
  | CHANGE [COLUMN] [IF EXISTS] old_col_name new_col_name column_definition
        [FIRST|AFTER col_name]
  | MODIFY [COLUMN] [IF EXISTS] col_name column_definition
        [FIRST | AFTER col_name]
  | DROP [COLUMN] [IF EXISTS] col_name [RESTRICT|CASCADE]
  | DROP PRIMARY KEY
  | DROP {INDEX|KEY} [IF EXISTS] index_name
  | DROP FOREIGN KEY [IF EXISTS] fk_symbol
  | DROP CONSTRAINT [IF EXISTS] constraint_name
  | DISABLE KEYS
  | ENABLE KEYS
  | RENAME [TO] new_tbl_name
  | ORDER BY col_name [, col_name] ...
  | RENAME COLUMN old_col_name TO new_col_name
  | RENAME {INDEX|KEY} old_index_name TO new_index_name
  | CONVERT TO CHARACTER SET charset_name [COLLATE collation_name]
  | [DEFAULT] CHARACTER SET [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
  | DISCARD TABLESPACE
  | IMPORT TABLESPACE
  | ALGORITHM [=] {DEFAULT|INPLACE|COPY|NOCOPY|INSTANT}
  | LOCK [=] {DEFAULT|NONE|SHARED|EXCLUSIVE}
  | FORCE
  | partition_options
  | ADD PARTITION [IF NOT EXISTS] (partition_definition)
  | DROP PARTITION [IF EXISTS] partition_names
  | COALESCE PARTITION number
  | REORGANIZE PARTITION [partition_names INTO (partition_definitions)]
  | ANALYZE PARTITION partition_names
  | CHECK PARTITION partition_names
  | OPTIMIZE PARTITION partition_names
  | REBUILD PARTITION partition_names
  | REPAIR PARTITION partition_names
```

```
  | EXCHANGE PARTITION partition_name WITH TABLE tbl_name
  | REMOVE PARTITIONING
  | ADD SYSTEM VERSIONING
  | DROP SYSTEM VERSIONING

index_col_name:
    col_name [(length)] [ASC | DESC]

index_type:
    USING {BTREE | HASH | RTREE}

index_option:
    [ KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
  | CLUSTERING={YES| NO} ]
  [ IGNORED | NOT IGNORED ]

table_options:
    table_option [[,] table_option] ...
```

Contents

Description

`ALTER TABLE` enables you to change the structure of an existing table. For example, you can add or delete columns, create or destroy indexes, change the type of existing columns, or rename columns or the table itself. You can also change the comment for the table and the storage engine of the table.

If another connection is using the table, a metadata lock is active, and this statement will wait until the lock is released. This is also true for

non-transactional tables.

When adding a `UNIQUE` index on a column (or a set of columns) which have duplicated values, an error will be produced and the statement will be stopped. To suppress the error and force the creation of `UNIQUE` indexes, discarding duplicates, the IGNORE option can be specified. This can be useful if a column (or a set of columns) should be UNIQUE but it contains duplicate values; however, this technique provides no control on which rows are preserved and which are deleted. Also, note that `IGNORE` is accepted but ignored in `ALTER TABLE ... EXCHANGE PARTITION` statements.

This statement can also be used to rename a table. For details see RENAME TABLE.

When an index is created, the storage engine may use a configurable buffer in the process. Incrementing the buffer speeds up the index creation. Aria and MyISAM allocate a buffer whose size is defined by aria_sort_buffer_size or myisam_sort_buffer_size, also used for REPAIR TABLE. InnoDB allocates three buffers whose size is defined by innodb_sort_buffer_size.

Privileges

Executing the `ALTER TABLE` statement generally requires at least the ALTER privilege for the table or the database..

If you are renaming a table, then it also requires the DROP, CREATE and INSERT privileges for the table or the database as well.

Online DDL

Online DDL is supported with the ALGORITHM and LOCK clauses.

See InnoDB Online DDL Overview for more information on online DDL with InnoDB.

ALTER ONLINE TABLE

ALTER ONLINE TABLE also works for partitioned tables.

Online `ALTER TABLE` is available by executing the following:

```
ALTER ONLINE TABLE ...;
```

This statement has the following semantics:

This statement is equivalent to the following:

```
ALTER TABLE ... LOCK=NONE;
```

See the LOCK alter specification for more information. <</product>>

This statement is equivalent to the following:

```
ALTER TABLE ... ALGORITHM=INPLACE;
```

See the ALGORITHM alter specification for more information. <</product>>

WAIT/NOWAIT

MariaDB starting with 10.3.0

Set the lock wait timeout. See WAIT and NOWAIT.

IF EXISTS

The `IF EXISTS` and `IF NOT EXISTS` clauses are available for the following:

```
ADD COLUMN       [IF NOT EXISTS]
ADD INDEX        [IF NOT EXISTS]
ADD FOREIGN KEY  [IF NOT EXISTS]
ADD PARTITION    [IF NOT EXISTS]
CREATE INDEX     [IF NOT EXISTS]

DROP COLUMN      [IF EXISTS]
DROP INDEX       [IF EXISTS]
DROP FOREIGN KEY [IF EXISTS]
DROP PARTITION   [IF EXISTS]
CHANGE COLUMN    [IF EXISTS]
MODIFY COLUMN    [IF EXISTS]
DROP INDEX       [IF EXISTS]
```

When `IF EXISTS` and `IF NOT EXISTS` are used in clauses, queries will not report errors when the condition is triggered for that clause. A warning with the same message text will be issued and the ALTER will move on to the next clause in the statement (or end if finished). <</product>>

MariaDB starting with 10.5.2

If this is directive is used after `ALTER ... TABLE`, one will not get an error if the table doesn't exist.

Column Definitions

See CREATE TABLE: Column Definitions for information about column definitions.

Index Definitions

See CREATE TABLE: Index Definitions for information about index definitions.

The CREATE INDEX and DROP INDEX statements can also be used to add or remove an index.

Character Sets and Collations

```
CONVERT TO CHARACTER SET charset_name [COLLATE collation_name]
[DEFAULT] CHARACTER SET [=] charset_name
[DEFAULT] COLLATE [=] collation_name
```

See Setting Character Sets and Collations for details on setting the character sets and collations.

Alter Specifications

Table Options

See CREATE TABLE: Table Options for information about table options.

ADD COLUMN

```
... ADD COLUMN [IF NOT EXISTS]  (col_name column_definition,...)
```

Adds a column to the table. The syntax is the same as in CREATE TABLE. If you are using `IF NOT_EXISTS` the column will not be added if it was not there already. This is very useful when doing scripts to modify tables.

The `FIRST` and `AFTER` clauses affect the physical order of columns in the datafile. Use `FIRST` to add a column in the first (leftmost) position, or `AFTER` followed by a column name to add the new column in any other position. Note that, nowadays, the physical position of a column is usually irrelevant.

See also Instant ADD COLUMN for InnoDB.

DROP COLUMN

```
... DROP COLUMN [IF EXISTS] col_name [CASCADE|RESTRICT]
```

Drops the column from the table. If you are using `IF EXISTS` you will not get an error if the column didn't exist. If the column is part of any index, the column will be dropped from them, except if you add a new column with identical name at the same time. The index will be dropped if all columns from the index were dropped. If the column was used in a view or trigger, you will get an error next time the view or trigger is accessed.

MariaDB starting with 10.2.8

Dropping a column that is part of a multi-column `UNIQUE` constraint is not permitted. For example:

```
CREATE TABLE a (
  a int,
  b int,
  primary key (a,b)
);

ALTER TABLE x DROP COLUMN a;
[42000][1072] Key column 'A' doesn't exist in table
```

The reason is that dropping column `a` would result in the new constraint that all values in column `b` be unique. In order to drop the column, an explicit `DROP PRIMARY KEY` and `ADD PRIMARY KEY` would be required. Up until MariaDB 10.2.7, the column was dropped and the additional constraint applied, resulting in the following structure:

```
ALTER TABLE x DROP COLUMN a;
Query OK, 0 rows affected (0.46 sec)

DESC x;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| b     | int(11) | NO   | PRI | NULL    |       |
+-------+---------+------+-----+---------+-------+
```

MariaDB starting with 10.4.0

MariaDB 10.4.0 supports instant DROP COLUMN. DROP COLUMN of an indexed column would imply DROP INDEX (and in the case of a non-UNIQUE multi-column index, possibly ADD INDEX). These will not be allowed with ALGORITHM=INSTANT, but unlike before, they can be allowed with ALGORITHM=NOCOPY

`RESTRICT` and `CASCADE` are allowed to make porting from other database systems easier. In MariaDB, they do nothing.

MODIFY COLUMN

Allows you to modify the type of a column. The column will be at the same place as the original column and all indexes on the column will be kept. Note that when modifying column, you should specify all attributes for the new column.

```
CREATE TABLE t1 (a INT UNSIGNED AUTO_INCREMENT, PRIMARY KEY((a));
ALTER TABLE t1 MODIFY a BIGINT UNSIGNED AUTO_INCREMENT;
```

CHANGE COLUMN

Works like `MODIFY COLUMN` except that you can also change the name of the column. The column will be at the same place as the original column and all index on the column will be kept.

```
CREATE TABLE t1 (a INT UNSIGNED AUTO_INCREMENT, PRIMARY KEY(a));
ALTER TABLE t1 CHANGE a b BIGINT UNSIGNED AUTO_INCREMENT;
```

ALTER COLUMN

This lets you change column options.

```
CREATE TABLE t1 (a INT UNSIGNED AUTO_INCREMENT, b varchar(50), PRIMARY KEY(a));
ALTER TABLE t1 ALTER b SET DEFAULT 'hello';
```

RENAME INDEX/KEY

MariaDB starting with 10.5.2

From MariaDB 10.5.2, it is possible to rename an index using the `RENAME INDEX` (or `RENAME KEY`) syntax, for example:

```
ALTER TABLE t1 RENAME INDEX i_old TO i_new;
```

RENAME COLUMN

MariaDB starting with 10.5.2

From MariaDB 10.5.2, it is possible to rename a column using the `RENAME COLUMN` syntax, for example:

```
ALTER TABLE t1 RENAME COLUMN c_old TO c_new;
```

ADD PRIMARY KEY

Add a primary key.

For `PRIMARY KEY` indexes, you can specify a name for the index, but it is silently ignored, and the name of the index is always `PRIMARY`.

See Getting Started with Indexes: Primary Key for more information.

DROP PRIMARY KEY

Drop a primary key.

For `PRIMARY KEY` indexes, you can specify a name for the index, but it is silently ignored, and the name of the index is always `PRIMARY`.

See Getting Started with Indexes: Primary Key for more information.

ADD FOREIGN KEY

Add a foreign key.

For `FOREIGN KEY` indexes, a reference definition must be provided.

For `FOREIGN KEY` indexes, you can specify a name for the constraint, using the `CONSTRAINT` keyword. That name will be used in error messages.

First, you have to specify the name of the target (parent) table and a column or a column list which must be indexed and whose values must match to the foreign key's values. The `MATCH` clause is accepted to improve the compatibility with other DBMS's, but has no meaning in MariaDB. The `ON DELETE` and `ON UPDATE` clauses specify what must be done when a `DELETE` (or a `REPLACE`) statements attempts to delete a referenced row from the parent table, and when an `UPDATE` statement attempts to modify the referenced foreign key columns in a parent table row, respectively. The following options are allowed:

- `RESTRICT`: The delete/update operation is not performed. The statement terminates with a 1451 error (SQLSTATE '2300').
- `NO ACTION`: Synonym for `RESTRICT`.
- `CASCADE`: The delete/update operation is performed in both tables.
- `SET NULL`: The update or delete goes ahead in the parent table, and the corresponding foreign key fields in the child table are set to `NULL`. (They must not be defined as `NOT NULL` for this to succeed).
- `SET DEFAULT`: This option is implemented only for the legacy PBXT storage engine, which is disabled by default and no longer maintained. It sets the child table's foreign key fields to their `DEFAULT` values when the referenced parent table key entries are updated or deleted.

If either clause is omitted, the default behavior for the omitted clause is `RESTRICT`.

See Foreign Keys for more information.

DROP FOREIGN KEY

Drop a foreign key.

See Foreign Keys for more information.

ADD INDEX

Add a plain index.

Plain indexes are regular indexes that are not unique, and are not acting as a primary key or a foreign key. They are also not the "specialized" `FULLTEXT` or `SPATIAL` indexes.

See Getting Started with Indexes: Plain Indexes for more information.

DROP INDEX

Drop a plain index.

Plain indexes are regular indexes that are not unique, and are not acting as a primary key or a foreign key. They are also not the "specialized" `FULLTEXT` or `SPATIAL` indexes.

See Getting Started with Indexes: Plain Indexes for more information.

ADD UNIQUE INDEX

Add a unique index.

The `UNIQUE` keyword means that the index will not accept duplicated values, except for NULLs. An error will raise if you try to insert duplicate values in a UNIQUE index.

For `UNIQUE` indexes, you can specify a name for the constraint, using the `CONSTRAINT` keyword. That name will be used in error messages.

See Getting Started with Indexes: Unique Index for more information.

DROP UNIQUE INDEX

Drop a unique index.

The `UNIQUE` keyword means that the index will not accept duplicated values, except for NULLs. An error will raise if you try to insert duplicate values in a UNIQUE index.

For `UNIQUE` indexes, you can specify a name for the constraint, using the `CONSTRAINT` keyword. That name will be used in error messages.

See Getting Started with Indexes: Unique Index for more information.

ADD FULLTEXT INDEX

Add a `FULLTEXT` index.

See Full-Text Indexes for more information.

DROP FULLTEXT INDEX

Drop a `FULLTEXT` index.

See Full-Text Indexes for more information.

ADD SPATIAL INDEX

Add a SPATIAL index.

See SPATIAL INDEX for more information.

DROP SPATIAL INDEX

Drop a SPATIAL index.

See SPATIAL INDEX for more information.

ENABLE/ DISABLE KEYS

`DISABLE KEYS` will disable all non unique keys for the table for storage engines that support this (at least MyISAM and Aria). This can be used

to speed up inserts into empty tables.

`ENABLE KEYS` will enable all disabled keys.

RENAME TO

Renames the table. See also RENAME TABLE.

ADD CONSTRAINT

Modifies the table adding a constraint on a particular column or columns.

> MariaDB starting with 10.2.1
>
> MariaDB 10.2.1 introduced new ways to define a constraint.

Note: Before MariaDB 10.2.1, constraint expressions were accepted in syntax, but ignored.

```
ALTER TABLE table_name
ADD CONSTRAINT [constraint_name] CHECK(expression);
```

Before a row is inserted or updated, all constraints are evaluated in the order they are defined. If any constraint fails, then the row will not be updated. One can use most deterministic functions in a constraint, including UDF's.

```
CREATE TABLE account_ledger (
 id INT PRIMARY KEY AUTO_INCREMENT,
 transaction_name VARCHAR(100),
 credit_account VARCHAR(100),
 credit_amount INT,
 debit_account VARCHAR(100),
 debit_amount INT);

ALTER TABLE account_ledger
ADD CONSTRAINT is_balanced
    CHECK((debit_amount + credit_amount) = 0);
```

The `constraint_name` is optional. If you don't provide one in the `ALTER TABLE` statement, MariaDB auto-generates a name for you. This is done so that you can remove it later using DROP CONSTRAINT clause.

You can disable all constraint expression checks by setting the variable check_constraint_checks to `OFF`. You may find this useful when loading a table that violates some constraints that you want to later find and fix in SQL.

To view constraints on a table, query information_schema.TABLE_CONSTRAINTS:

```
SELECT CONSTRAINT_NAME, TABLE_NAME, CONSTRAINT_TYPE
FROM information_schema.TABLE_CONSTRAINTS
WHERE TABLE_NAME = 'account_ledger';

+-----------------+----------------+-----------------+
| CONSTRAINT_NAME | TABLE_NAME     | CONSTRAINT_TYPE |
+-----------------+----------------+-----------------+
| is_balanced     | account_ledger | CHECK           |
+-----------------+----------------+-----------------+
```

DROP CONSTRAINT

> MariaDB starting with 10.2.22
>
> `DROP CONSTRAINT` for `UNIQUE` and `FOREIGN KEY` constraints was introduced in MariaDB 10.2.22 and MariaDB 10.3.13.

> MariaDB starting with 10.2.1
>
> `DROP CONSTRAINT` for `CHECK` constraints was introduced in MariaDB 10.2.1

Modifies the table, removing the given constraint.

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

When you add a constraint to a table, whether through a CREATE TABLE or ALTER TABLE...ADD CONSTRAINT statement, you can either set a `constraint_name` yourself, or allow MariaDB to auto-generate one for you. To view constraints on a table, query information_schema.TABLE_CONSTRAINTS. For instance,

```
CREATE TABLE t (
   a INT,
   b INT,
   c INT,
   CONSTRAINT CHECK(a > b),
   CONSTRAINT check_equals CHECK(a = c));

SELECT CONSTRAINT_NAME, TABLE_NAME, CONSTRAINT_TYPE
FROM information_schema.TABLE_CONSTRAINTS
WHERE TABLE_NAME = 't';

+-----------------+----------------+-----------------+
| CONSTRAINT_NAME | TABLE_NAME     | CONSTRAINT_TYPE |
+-----------------+----------------+-----------------+
| check_equals    | t              | CHECK           |
| CONSTRAINT_1    | t              | CHECK           |
+-----------------+----------------+-----------------+
```

To remove a constraint from the table, issue an `ALTER TABLE...DROP CONSTRAINT` statement. For example,

```
ALTER TABLE t DROP CONSTRAINT is_unique;
```

ADD SYSTEM VERSIONING

MariaDB starting with 10.3.4

System-versioned tables was added in MariaDB 10.3.4.

Add system versioning.

DROP SYSTEM VERSIONING

MariaDB starting with 10.3.4

System-versioned tables was added in MariaDB 10.3.4.

Drop system versioning.

ADD PERIOD FOR SYSTEM_TIME

MariaDB starting with 10.3.4

System-versioned tables was added in MariaDB 10.3.4.

FORCE

`ALTER TABLE ... FORCE` can force MariaDB to re-build the table.

In MariaDB 5.5 and before, this could only be done by setting the ENGINE table option to its old value. For example, for an InnoDB table, one could execute the following:

```
ALTER TABLE tab_name ENGINE = InnoDB;
```

The `FORCE` option can be used instead. For example, :

```
ALTER TABLE tab_name FORCE;
```

With InnoDB, the table rebuild will only reclaim unused space (i.e. the space previously used for deleted rows) if the innodb_file_per_table system variable is set to `ON`. If the system variable is `OFF`, then the space will not be reclaimed, but it will be-re-used for new data that's later added.

EXCHANGE PARTITION

This is used to exchange the tablespace files between a partition and another table.

See copying InnoDB's transportable tablespaces for more information.

DISCARD TABLESPACE

This is used to discard an InnoDB table's tablespace.

See copying InnoDB's transportable tablespaces for more information.

IMPORT TABLESPACE

This is used to import an InnoDB table's tablespace. The tablespace should have been copied from its original server after executing FLUSH TABLES FOR EXPORT.

See copying InnoDB's transportable tablespaces for more information.

`ALTER TABLE ... IMPORT` only applies to InnoDB tables. Most other popular storage engines, such as Aria and MyISAM, will recognize their data files as soon as they've been placed in the proper directory under the datadir, and no special DDL is required to import them.

ALGORITHM

The `ALTER TABLE` statement supports the `ALGORITHM` clause. This clause is one of the clauses that is used to implement online DDL. `ALTER TABLE` supports several different algorithms. An algorithm can be explicitly chosen for an `ALTER TABLE` operation by setting the `ALGORITHM` clause. The supported values are:

- `ALGORITHM=DEFAULT` - This implies the default behavior for the specific statement, such as if no `ALGORITHM` clause is specified.
- `ALGORITHM=COPY`
- `ALGORITHM=INPLACE`
- `ALGORITHM=NOCOPY` - This was added in MariaDB 10.3.7.
- `ALGORITHM=INSTANT` - This was added in MariaDB 10.3.7.

See InnoDB Online DDL Overview: ALGORITHM for information on how the `ALGORITHM` clause affects InnoDB.

ALGORITHM=DEFAULT

The default behavior, which occurs if `ALGORITHM=DEFAULT` is specified, or if `ALGORITHM` is not specified at all, usually only makes a copy if the operation doesn't support being done in-place at all. In this case, the most efficient available algorithm will usually be used.

However, in MariaDB 10.3.6 and before, if the value of the old_alter_table system variable is set to `ON`, then the default behavior is to perform `ALTER TABLE` operations by making a copy of the table using the old algorithm.

In MariaDB 10.3.7 and later, the old_alter_table system variable is deprecated. Instead, the alter_algorithm system variable defines the default algorithm for `ALTER TABLE` operations.

ALGORITHM=COPY

`ALGORITHM=COPY` is the name for the original ALTER TABLE algorithm from early MariaDB versions.

When `ALGORITHM=COPY` is set, MariaDB essentially does the following operations:

```
-- Create a temporary table with the new definition
CREATE TEMPORARY TABLE tmp_tab (
...
);

-- Copy the data from the original table
INSERT INTO tmp_tab
   SELECT * FROM original_tab;

-- Drop the original table
DROP TABLE original_tab;

-- Rename the temporary table, so that it replaces the original one
RENAME TABLE tmp_tab TO original_tab;
```

This algorithm is very inefficient, but it is generic, so it works for all storage engines.

If `ALGORITHM=COPY` is specified, then the copy algorithm will be used even if it is not necessary. This can result in a lengthy table copy. If multiple ALTER TABLE operations are required that each require the table to be rebuilt, then it is best to specify all operations in a single ALTER TABLE statement, so that the table is only rebuilt once.

ALGORITHM=INPLACE

`ALGORITHM=COPY` can be incredibly slow, because the whole table has to be copied and rebuilt. `ALGORITHM=INPLACE` was introduced as a way to avoid this by performing operations in-place and avoiding the table copy and rebuild, when possible.

When `ALGORITHM=INPLACE` is set, the underlying storage engine uses optimizations to perform the operation while avoiding the table copy and rebuild. However, `INPLACE` is a bit of a misnomer, since some operations may still require the table to be rebuilt for some storage engines. Regardless, several operations can be performed without a full copy of the table for some storage engines.

A more accurate name would have been `ALGORITHM=ENGINE`, where `ENGINE` refers to an "engine-specific" algorithm.

If an ALTER TABLE operation supports `ALGORITHM=INPLACE`, then it can be performed using optimizations by the underlying storage engine, but it may rebuilt.

See InnoDB Online DDL Operations with ALGORITHM=INPLACE for more.

ALGORITHM=NOCOPY

`ALGORITHM=NOCOPY` was introduced in MariaDB 10.3.7.

`ALGORITHM=INPLACE` can sometimes be surprisingly slow in instances where it has to rebuild the clustered index, because when the clustered index has to be rebuilt, the whole table has to be rebuilt. `ALGORITHM=NOCOPY` was introduced as a way to avoid this.

If an `ALTER TABLE` operation supports `ALGORITHM=NOCOPY`, then it can be performed without rebuilding the clustered index.

If `ALGORITHM=NOCOPY` is specified for an `ALTER TABLE` operation that does not support `ALGORITHM=NOCOPY`, then an error will be raised. In this case, raising an error is preferable, if the alternative is for the operation to rebuild the clustered index, and perform unexpectedly slowly.

See InnoDB Online DDL Operations with ALGORITHM=NOCOPY for more.

ALGORITHM=INSTANT

`ALGORITHM=INSTANT` was introduced in MariaDB 10.3.7.

`ALGORITHM=INPLACE` can sometimes be surprisingly slow in instances where it has to modify data files. `ALGORITHM=INSTANT` was introduced as a way to avoid this.

If an `ALTER TABLE` operation supports `ALGORITHM=INSTANT`, then it can be performed without modifying any data files.

If `ALGORITHM=INSTANT` is specified for an `ALTER TABLE` operation that does not support `ALGORITHM=INSTANT`, then an error will be raised. In this case, raising an error is preferable, if the alternative is for the operation to modify data files, and perform unexpectedly slowly.

See InnoDB Online DDL Operations with ALGORITHM=INSTANT for more.

LOCK

The `ALTER TABLE` statement supports the `LOCK` clause. This clause is one of the clauses that is used to implement online DDL. `ALTER TABLE` supports several different locking strategies. A locking strategy can be explicitly chosen for an `ALTER TABLE` operation by setting the `LOCK` clause. The supported values are:

- `DEFAULT`: Acquire the least restrictive lock on the table that is supported for the specific operation. Permit the maximum amount of concurrency that is supported for the specific operation.
- `NONE`: Acquire no lock on the table. Permit **all** concurrent DML. If this locking strategy is not permitted for an operation, then an error is raised.
- `SHARED`: Acquire a read lock on the table. Permit **read-only** concurrent DML. If this locking strategy is not permitted for an operation, then an error is raised.
- `EXCLUSIVE`: Acquire a write lock on the table. Do **not** permit concurrent DML.

Different storage engines support different locking strategies for different operations. If a specific locking strategy is chosen for an `ALTER TABLE` operation, and that table's storage engine does not support that locking strategy for that specific operation, then an error will be raised.

If the `LOCK` clause is not explicitly set, then the operation uses `LOCK=DEFAULT`.

ALTER ONLINE TABLE is equivalent to `LOCK=NONE`. Therefore, the ALTER ONLINE TABLE statement can be used to ensure that your `ALTER TABLE` operation allows all concurrent DML.

See InnoDB Online DDL Overview: LOCK for information on how the `LOCK` clause affects InnoDB.

Progress Reporting

MariaDB provides progress reporting for `ALTER TABLE` statement for clients that support the new progress reporting protocol. For example, if you were using the mysql client, then the progress report might look like this::

```
ALTER TABLE test ENGINE=Aria;
Stage: 1 of 2 'copy to tmp table'    46% of stage
```

The progress report is also shown in the output of the SHOW PROCESSLIST statement and in the contents of the information_schema.PROCESSLIST table.

See Progress Reporting for more information.

Aborting ALTER TABLE Operations

If an `ALTER TABLE` operation is being performed and the connection is killed, the changes will be rolled back in a controlled manner. The rollback can be a slow operation as the time it takes is relative to how far the operation has progressed.

MariaDB starting with 10.2.13

Aborting `ALTER TABLE ... ALGORITHM=COPY` was made faster by removing excessive undo logging (MDEV-11415). This significantly shortens the time it takes to abort a running ALTER TABLE operation.

Atomic ALTER TABLE

MariaDB starting with 10.6.1

From MariaDB 10.6, `ALTER TABLE` is atomic for most engines, including InnoDB, MyRocks, MyISAM and Aria (MDEV-25180). This means that if there is a crash (server down or power outage) during an `ALTER TABLE` operation, after recovery, either the old table and associated triggers and status will be intact, or the new table will be active.

In older MariaDB versions one could get leftover #sql-alter..', '#sql-backup..' or 'table_name.frm˜' files if the system crashed during the `ALTER TABLE` operation.

See Atomic DDL for more information.

Replication

MariaDB starting with 10.8.0

Before MariaDB 10.8.0, ALTER TABLE got fully executed on the primary first, and only then was it replicated and started executing on replicas. From MariaDB 10.8.0, ALTER TABLE gets replicated and starts executing on replicas when it *starts* executing on the primary, not when it *finishes*. This way the replication lag caused by a heavy ALTER TABLE can be completely eliminated (MDEV-11675).

Examples

Adding a new column:

```
ALTER TABLE t1 ADD x INT;
```

Dropping a column:

```
ALTER TABLE t1 DROP x;
```

Modifying the type of a column:

```
ALTER TABLE t1 MODIFY x bigint unsigned;
```

Changing the name and type of a column:

```
ALTER TABLE t1 CHANGE a b bigint unsigned auto_increment;
```

Combining multiple clauses in a single ALTER TABLE statement, separated by commas:

```
ALTER TABLE t1 DROP x, ADD x2 INT,  CHANGE y y2 INT;
```

Changing the storage engine and adding a comment:

```
ALTER TABLE t1
  ENGINE = InnoDB
  COMMENT = 'First of three tables containing usage info';
```

Rebuilding the table (the previous example will also rebuild the table if it was already InnoDB):

```
ALTER TABLE t1 FORCE;
```

Dropping an index:

```
ALTER TABLE rooms DROP INDEX u;
```

Adding a unique index:

```
ALTER TABLE rooms ADD UNIQUE INDEX u(room_number);
```

From MariaDB 10.5.3, adding a primary key for an application-time period table with a WITHOUT OVERLAPS constraint:

```
ALTER TABLE rooms ADD PRIMARY KEY(room_number, p WITHOUT OVERLAPS);
```

See Also

- CREATE TABLE
- DROP TABLE
- Character Sets and Collations
- SHOW CREATE TABLE
- Instant ADD COLUMN for InnoDB

1.1.2.1.1.2 ALTER DATABASE

Modifies a database, changing its overall characteristics.

Syntax

```
ALTER {DATABASE | SCHEMA} [db_name]
    alter_specification ...
ALTER {DATABASE | SCHEMA} db_name
    UPGRADE DATA DIRECTORY NAME

alter_specification:
    [DEFAULT] CHARACTER SET [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
  | COMMENT [=] 'comment'
```

Contents

Description

`ALTER DATABASE` enables you to change the overall characteristics of a database. These characteristics are stored in the `db.opt` file in the database directory. To use `ALTER DATABASE`, you need the `ALTER` privilege on the database. `ALTER SCHEMA` is a synonym for ALTER DATABASE.

The `CHARACTER SET` clause changes the default database character set. The `COLLATE` clause changes the default database collation. See Character Sets and Collations for more.

You can see what character sets and collations are available using, respectively, the SHOW CHARACTER SET and SHOW COLLATION statements.

Changing the default character set/collation of a database does not change the character set/collation of any stored procedures or stored functions that were previously created, and relied on the defaults. These need to be dropped and recreated in order to apply the character set/collation changes.

The database name can be omitted from the first syntax, in which case the statement applies to the default database.

The syntax that includes the `UPGRADE DATA DIRECTORY NAME` clause was added in MySQL 5.1.23. It updates the name of the directory associated with the database to use the encoding implemented in MySQL 5.1 for mapping database names to database directory names (see Identifier to File Name Mapping). This clause is for use under these conditions:

- It is intended when upgrading MySQL to 5.1 or later from older versions.
- It is intended to update a database directory name to the current encoding format if the name contains special characters that need encoding.
- The statement is used by `mysqlcheck` (as invoked by `mysql_upgrade`).

For example,if a database in MySQL 5.0 has a name of a-b-c, the name contains instance of the `-' character. In 5.0, the database directory is also named a-b-c, which is not necessarily safe for all file systems. In MySQL 5.1 and up, the same database name is encoded as a@002db@002dc to produce a file system-neutral directory name.

When a MySQL installation is upgraded to MySQL 5.1 or later from an older version,the server displays a name such as a-b-c (which is in the old format) as #mysql50#a-b-c, and you must refer to the name using the #mysql50# prefix. Use `UPGRADE DATA DIRECTORY NAME` in this case to explicitly tell the server to re-encode the database directory name to the current encoding format:

```
ALTER DATABASE `#mysql50#a-b-c` UPGRADE DATA DIRECTORY NAME;
```

After executing this statement, you can refer to the database as a-b-c without the special #mysql50# prefix.

COMMENT

MariaDB starting with 10.5.0

From MariaDB 10.5.0, it is possible to add a comment of a maximum of 1024 bytes. If the comment length exceeds this length, a error/warning code 4144 is thrown. The database comment is also added to the db.opt file, as well as to the information_schema.schemata table.

Examples

```
ALTER DATABASE test CHARACTER SET='utf8'  COLLATE='utf8_bin';
```

From MariaDB 10.5.0:

```
ALTER DATABASE p COMMENT='Presentations';
```

See Also

- CREATE DATABASE
- DROP DATABASE
- SHOW CREATE DATABASE
- SHOW DATABASES
- Character Sets and Collations
- Information Schema SCHEMATA Table

1.1.2.1.1.3 ALTER EVENT

Modifies one or more characteristics of an existing event.

Syntax

```
ALTER
    [DEFINER = { user | CURRENT_USER }]
    EVENT event_name
    [ON SCHEDULE schedule]
    [ON COMPLETION [NOT] PRESERVE]
    [RENAME TO new_event_name]
    [ENABLE | DISABLE | DISABLE ON SLAVE]
    [COMMENT 'comment']
    [DO sql_statement]
```

Contents

Description

The `ALTER EVENT` statement is used to change one or more of the characteristics of an existing event without the need to drop and recreate it. The syntax for each of the `DEFINER`, `ON SCHEDULE`, `ON COMPLETION`, `COMMENT`, `ENABLE / DISABLE`, and `DO` clauses is exactly the same as when used with `CREATE EVENT`.

This statement requires the `EVENT` privilege. When a user executes a successful `ALTER EVENT` statement, that user becomes the definer for the affected event.

(In MySQL 5.1.11 and earlier, an event could be altered only by its definer, or by a user having the `SUPER` privilege.)

`ALTER EVENT` works only with an existing event:

```
ALTER EVENT no_such_event ON SCHEDULE EVERY '2:3' DAY_HOUR;
ERROR 1539 (HY000): Unknown event 'no_such_event'
```

Examples

```
ALTER EVENT myevent
  ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 2 HOUR
  DO
    UPDATE myschema.mytable SET mycol = mycol + 1;
```

See Also

- Events Overview
- CREATE EVENT
- SHOW CREATE EVENT
- DROP EVENT

1.1.2.1.1.4 ALTER FUNCTION

Syntax

```
ALTER FUNCTION func_name [characteristic ...]

characteristic:
    { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'
```

Contents

Description

This statement can be used to change the characteristics of a stored function. More than one change may be specified in an `ALTER FUNCTION` statement. However, you cannot change the parameters or body of a stored function using this statement; to make such changes, you must drop and re-create the function using DROP FUNCTION and CREATE FUNCTION.

You must have the `ALTER ROUTINE` privilege for the function. (That privilege is granted automatically to the function creator.) If binary logging is enabled, the `ALTER FUNCTION` statement might also require the `SUPER` privilege, as described in Binary Logging of Stored Routines.

Example

```
ALTER FUNCTION hello SQL SECURITY INVOKER;
```

See Also

- CREATE FUNCTION
- SHOW CREATE FUNCTION
- DROP FUNCTION
- SHOW FUNCTION STATUS
- Information Schema ROUTINES Table

1.1.2.1.1.5 ALTER LOGFILE GROUP

Syntax

```
ALTER LOGFILE GROUP logfile_group
    ADD UNDOFILE 'file_name'
    [INITIAL_SIZE [=] size]
    [WAIT]
    ENGINE [=] engine_name
```

The `ALTER LOGFILE GROUP` statement is not supported by MariaDB. It was originally inherited from MySQL NDB Cluster. See MDEV-19295 for more information.

1.1.2.1.1.6 ALTER PROCEDURE

Syntax

```
ALTER PROCEDURE proc_name [characteristic ...]

characteristic:
    { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'
```

Description

This statement can be used to change the characteristics of a stored procedure. More than one change may be specified in an `ALTER PROCEDURE` statement. However, you cannot change the parameters or body of a stored procedure using this statement. To make such changes, you must drop and re-create the procedure using either CREATE OR REPLACE PROCEDURE (since MariaDB 10.1.3) or DROP PROCEDURE and CREATE PROCEDURE (MariaDB 10.1.2 and before).

You must have the `ALTER ROUTINE` privilege for the procedure. By default, that privilege is granted automatically to the procedure creator. See Stored Routine Privileges.

Example

```
ALTER PROCEDURE simpleproc SQL SECURITY INVOKER;
```

See Also

- Stored Procedure Overview
- CREATE PROCEDURE
- SHOW CREATE PROCEDURE
- DROP PROCEDURE
- SHOW CREATE PROCEDURE
- SHOW PROCEDURE STATUS
- Stored Routine Privileges
- Information Schema ROUTINES Table

1.1.2.1.1.7 ALTER SEQUENCE

MariaDB starting with 10.3.1

`ALTER SEQUENCE` was introduced in MariaDB 10.3.

Syntax

```
ALTER SEQUENCE [IF EXISTS] sequence_name
[ INCREMENT [ BY | = ] increment ]
[ MINVALUE [=] minvalue | NO MINVALUE | NOMINVALUE ]
[ MAXVALUE [=] maxvalue | NO MAXVALUE | NOMAXVALUE ]
[ START [ WITH | = ] start ] [ CACHE [=] cache ] [ [ NO ] CYCLE ]
[ RESTART [[WITH | =] restart]
```

Contents

`ALTER SEQUENCE` allows one to change any values for a `SEQUENCE` created with CREATE SEQUENCE.

The options for `ALTER SEQUENCE` can be given in any order.

Description

`ALTER SEQUENCE` changes the parameters of an existing sequence generator. Any parameters not specifically set in the `ALTER SEQUENCE` command retain their prior settings.

`ALTER SEQUENCE` requires the ALTER privilege.

Arguments to `ALTER SEQUENCE`

The following options may be used:

Option	Default value	Description
`INCREMENT`	1	Increment to use for values. May be negative.
`MINVALUE`	1 if `INCREMENT` > 0 and -9223372036854775807 if `INCREMENT` < 0	Minimum value for the sequence.
`MAXVALUE`	9223372036854775806 if `INCREMENT` > 0 and -1 if `INCREMENT` < 0	Max value for sequence.
`START`	`MINVALUE` if `INCREMENT` > 0 and `MAX_VALUE` if `INCREMENT` < 0	First value that the sequence will generate.
`CACHE`	1000	Number of values that should be cached. 0 if no `CACHE` . The underlying table will be updated first time a new sequence number is generated and each time the cache runs out.
`CYCLE`	0 (= `NO CYCLE`)	1 if the sequence should start again from `MINVALUE` # after it has run out of values.
`RESTART`	`START` if `restart` value not is given	If `RESTART` option is used, `NEXT VALUE` will return the restart value.

The optional clause `RESTART [ WITH restart ]` sets the next value for the sequence. This is equivalent to calling the SETVAL() function with the `is_used` argument as 0. The specified value will be returned by the next call of nextval. Using `RESTART` with no restart value is equivalent to supplying the start value that was recorded by CREATE SEQUENCE or last set by `ALTER SEQUENCE START WITH` .

`ALTER SEQUENCE` will not allow you to change the sequence so that it's inconsistent. For example:

```
CREATE SEQUENCE s1;
ALTER SEQUENCE s1 MINVALUE 10;
ERROR 4061 (HY000): Sequence 'test.t1' values are conflicting

ALTER SEQUENCE s1 MINVALUE 10 RESTART 10;
ERROR 4061 (HY000): Sequence 'test.t1' values are conflicting

ALTER SEQUENCE s1 MINVALUE 10 START 10 RESTART 10;
```

INSERT

To allow `SEQUENCE` objects to be backed up by old tools, like mysqldump, one can use `SELECT` to read the current state of a `SEQUENCE` object and use an `INSERT` to update the `SEQUENCE` object. `INSERT` is only allowed if all fields are specified:

```
CREATE SEQUENCE s1;
INSERT INTO s1 VALUES(1000,10,2000,1005,1,1000,0,0);
SELECT * FROM s1;

+------------+-----------+-----------+-------+-----------+-------+-------+-------+
| next_value | min_value | max_value | start | increment | cache | cycle | round |
+------------+-----------+-----------+-------+-----------+-------+-------+-------+
|       1000 |        10 |      2000 |  1005 |         1 |  1000 |     0 |     0 |
+------------+-----------+-----------+-------+-----------+-------+-------+-------+

SHOW CREATE SEQUENCE s1;
+-------+-----------------------------------------------------------------------------------------------------------------+
| Table | Create Table                                                                                                    |
+-------+-----------------------------------------------------------------------------------------------------------------+
| s1    | CREATE SEQUENCE `s1` start with 1005 minvalue 10 maxvalue 2000 increment by 1 cache 1000 nocycle ENGINE=Aria |
+-------+-----------------------------------------------------------------------------------------------------------------+
```

Notes

`ALTER SEQUENCE` will instantly affect all future `SEQUENCE` operations. This is in contrast to some other databases where the changes requested by `ALTER SEQUENCE` will not be seen until the sequence cache has run out.

`ALTER SEQUENCE` will take a full table lock of the sequence object during its (brief) operation. This ensures that `ALTER SEQUENCE` is replicated correctly. If you only want to set the next sequence value to a higher value than current, then you should use SETVAL() instead, as this is not blocking.

If you want to change storage engine, sequence comment or rename the sequence, you can use ALTER TABLE for this.

See Also

- Sequence Overview
- CREATE SEQUENCE
- DROP SEQUENCE
- NEXT VALUE FOR
- PREVIOUS VALUE FOR
- SETVAL(). Set next value for the sequence.
- AUTO INCREMENT
- ALTER TABLE

1.1.2.1.1.8 ALTER SERVER

Syntax

```
ALTER SERVER server_name
    OPTIONS (option [, option] ...)
```

Contents

Description

Alters the server information for *server_name*, adjusting the specified options as per the CREATE SERVER command. The corresponding fields in the mysql.servers table are updated accordingly. This statement requires the SUPER privilege or, from MariaDB 10.5.2, the FEDERATED ADMIN privilege.

ALTER SERVER is not written to the binary log, irrespective of the binary log format being used. From MariaDB 10.1.13, Galera replicates the CREATE SERVER, ALTER SERVER and DROP SERVER statements.

Examples

```
ALTER SERVER s OPTIONS (USER 'sally');
```

See Also

- CREATE SERVER
- DROP SERVER
- Spider Storage Engine

1.1.2.1.1.9 ALTER TABLESPACE

The `ALTER TABLESPACE` statement is not supported by MariaDB. It was originally inherited from MySQL NDB Cluster. In MySQL 5.7 and later, the statement is also supported for InnoDB. However, MariaDB has chosen not to include that specific feature. See MDEV-19294 for more information.

1.1.2.1.1.10 ALTER USER

1.1.2.1.2 ALTER VIEW

Syntax

```
ALTER
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = { user | CURRENT_USER }]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

Contents

Description

This statement changes the definition of a view, which must exist. The syntax is similar to that for CREATE VIEW and the effect is the same as for `CREATE OR REPLACE VIEW` if the view exists. This statement requires the `CREATE VIEW` and `DROP` privileges for the view, and some privilege for each column referred to in the `SELECT` statement. `ALTER VIEW` is allowed only to the definer or users with the SUPER privilege.

Example

```
ALTER VIEW v AS SELECT a, a*3 AS a2 FROM t;
```

See Also

- CREATE VIEW
- DROP VIEW

- SHOW CREATE VIEW
- INFORMATION SCHEMA VIEWS Table

1.1.2.1.3 ANALYZE TABLE

Syntax

```
ANALYZE [NO_WRITE_TO_BINLOG | LOCAL] TABLE tbl_name [,tbl_name ...]
  [PERSISTENT FOR [ALL|COLUMNS ([col_name [,col_name ...]])]
    [INDEXES ([index_name [,index_name ...]])]]
```

Contents

Description

`ANALYZE TABLE` analyzes and stores the key distribution for a table (index statistics). This statement works with MyISAM, Aria and InnoDB tables. During the analysis, InnoDB will allow reads/writes, and MyISAM/Aria reads/inserts. For MyISAM tables, this statement is equivalent to using myisamchk --analyze.

For more information on how the analysis works within InnoDB, see InnoDB Limitations.

MariaDB uses the stored key distribution to decide the order in which tables should be joined when you perform a join 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.

This statement requires SELECT and INSERT privileges for the table.

By default, ANALYZE TABLE statements are written to the binary log and will be replicated. The `NO_WRITE_TO_BINLOG` keyword (`LOCAL` is an alias) will ensure the statement is not written to the binary log.

From MariaDB 10.3.19, `ANALYZE TABLE` statements are not logged to the binary log if read_only is set. See also Read-Only Replicas.

`ANALYZE TABLE` is also supported for partitioned tables. You can use `ALTER TABLE ... ANALYZE PARTITION` to analyze one or more partitions.

The Aria storage engine supports progress reporting for the `ANALYZE TABLE` statement.

Engine-Independent Statistics

`ANALYZE TABLE` supports engine-independent statistics. See Engine-Independent Table Statistics: Collecting Statistics with the ANALYZE TABLE Statement for more information.

See Also

- Index Statistics
- InnoDB Persistent Statistics
- Progress Reporting
- Engine-independent Statistics
- Histogram-based Statistics
- ANALYZE Statement

1.1.2.1.4 CHECK TABLE

Syntax

```
CHECK TABLE tbl_name [, tbl_name] ... [option] ...

option = {FOR UPGRADE | QUICK | FAST | MEDIUM | EXTENDED | CHANGED}
```

Description

`CHECK TABLE` checks a table or tables for errors. `CHECK TABLE` works for Archive, Aria, CSV, InnoDB, and MyISAM tables. For Aria and MyISAM tables, the key statistics are updated as well. For CSV, see also Checking and Repairing CSV Tables.

As an alternative, myisamchk is a commandline tool for checking MyISAM tables when the tables are not being accessed.

For checking dynamic columns integrity, `COLUMN_CHECK()` can be used.

`CHECK TABLE` can also check views for problems, such as tables that are referenced in the view definition that no longer exist.

`CHECK TABLE` is also supported for partitioned tables. You can use `ALTER TABLE ... CHECK PARTITION` to check one or more partitions.

The meaning of the different options are as follows - note that this can vary a bit between storage engines:

FOR UPGRADE	Do a very quick check if the storage format for the table has changed so that one needs to do a REPAIR. This is only needed when one upgrades between major versions of MariaDB or MySQL. This is usually done by running mysql_upgrade.
FAST	Only check tables that has not been closed properly or are marked as corrupt. Only supported by the MyISAM and Aria engines. For other engines the table is checked normally
CHANGED	Check only tables that has changed since last REPAIR / CHECK. Only supported by the MyISAM and Aria engines. For other engines the table is checked normally.
QUICK	Do a fast check. For MyISAM and Aria engine this means we skip checking the delete link chain which may take some time.
MEDIUM	Scan also the data files. Checks integrity between data and index files with checksums. In most cases this should find all possible errors.
EXTENDED	Does a full check to verify every possible error. For MyISAM and Aria we verify for each row that all it keys exists and points to the row. This may take a long time on big tables!

For most cases running `CHECK TABLE` without options or `MEDIUM` should be good enough.

The Aria storage engine supports progress reporting for this statement.

If you want to know if two tables are identical, take a look at `CHECKSUM TABLE` .

InnoDB

If `CHECK TABLE` finds an error in an InnoDB table, MariaDB might shutdown to prevent the error propagation. In this case, the problem will be reported in the error log. Otherwise the table or an index might be marked as corrupted, to prevent use. This does not happen with some minor problems, like a wrong number of entries in a secondary index. Those problems are reported in the output of `CHECK TABLE` .

Each tablespace contains a header with metadata. This header is not checked by this statement.

During the execution of `CHECK TABLE` , other threads may be blocked.

1.1.2.1.5 CHECK VIEW

Syntax

```
CHECK VIEW view_name
```

Description

The `CHECK VIEW` statement was introduced in MariaDB 10.0.18 to assist with fixing MDEV-6916, an issue introduced in MariaDB 5.2 where the view algorithms were swapped. It checks whether the view algorithm is correct. It is run as part of mysql_upgrade, and should not normally be required in regular use.

See Also

- REPAIR VIEW

1.1.2.1.6 CHECKSUM TABLE

Syntax

```
CHECKSUM TABLE tbl_name [, tbl_name] ... [ QUICK | EXTENDED ]
```

Contents

Description

`CHECKSUM TABLE` reports a table checksum. This is very useful if you want to know if two tables are the same (for example on a master and slave).

With `QUICK`, the live table checksum is reported if it is available, or `NULL` otherwise. This is very fast. A live checksum is enabled by specifying the `CHECKSUM=1` table option when you create the table; currently, this is supported only for Aria and MyISAM tables.

With `EXTENDED`, the entire table is read row by row and the checksum is calculated. This can be very slow for large tables.

If neither `QUICK` nor `EXTENDED` is specified, MariaDB returns a live checksum if the table storage engine supports it and scans the table otherwise.

`CHECKSUM TABLE` requires the SELECT privilege for the table.

For a nonexistent table, `CHECKSUM TABLE` returns `NULL` and generates a warning.

The table row format affects the checksum value. If the row format changes, the checksum will change. This means that when a table created with a MariaDB/MySQL version is upgraded to another version, the checksum value will probably change.

Two identical tables should always match to the same checksum value; however, also for non-identical tables there is a very slight chance that they will return the same value as the hashing algorithm is not completely collision-free.

Differences Between MariaDB and MySQL

`CHECKSUM TABLE` may give a different result as MariaDB doesn't ignore `NULL`s in the columns as MySQL 5.1 does (Later MySQL versions should calculate checksums the same way as MariaDB). You can get the 'old style' checksum in MariaDB by starting mysqld with the `--old` option. Note however that that the MyISAM and Aria storage engines in MariaDB are using the new checksum internally, so if you are using `--old`, the `CHECKSUM` command will be slower as it needs to calculate the checksum row by row. Starting from MariaDB Server 10.9, `--old` is deprecated and will be removed in a future release. Set `--old-mode` or `OLD_MODE` to `COMPAT_5_1_CHECKSUM` to get 'old style' checksum.

1.1.2.1.7 CREATE TABLE

Syntax

```
CREATE [OR REPLACE] [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...) [table_options    ]... [partition_options]
CREATE [OR REPLACE] [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [(create_definition,...)] [table_options    ]... [partition_options]
    select_statement
CREATE [OR REPLACE] [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
   { LIKE old_table_name | (LIKE old_table_name) }

select_statement:
    [IGNORE | REPLACE] [AS] SELECT ...   (Some legal select statement)
```

Contents

Description

Use the `CREATE TABLE` statement to create a table with the given name.

In its most basic form, the `CREATE TABLE` statement provides a table name followed by a list of columns, indexes, and constraints. By default, the table is created in the default database. Specify a database with *`db_name.tbl_name`*. If you quote the table name, you must quote the database name and table name separately as *`` `db_name`.`tbl_name` ``*. This is particularly useful for CREATE TABLE ... SELECT, because it allows to create a table into a database, which contains data from other databases. See Identifier Qualifiers.

If a table with the same name exists, error 1050 results. Use IF NOT EXISTS to suppress this error and issue a note instead. Use SHOW WARNINGS to see notes.

The `CREATE TABLE` statement automatically commits the current transaction, except when using the TEMPORARY keyword.

For valid identifiers to use as table names, see Identifier Names.

Note: if the default_storage_engine is set to ColumnStore then it needs setting on all UMs. Otherwise when the tables using the default engine are replicated across UMs they will use the wrong engine. You should therefore not use this option as a session variable with ColumnStore.

Microsecond precision can be between 0-6. If no precision is specified it is assumed to be 0, for backward compatibility reasons.

Privileges

Executing the `CREATE TABLE` statement requires the CREATE privilege for the table or the database.

CREATE OR REPLACE

If the `OR REPLACE` clause is used and the table already exists, then instead of returning an error, the server will drop the existing table and replace it with the newly defined table.

This syntax was originally added to make replication more robust if it has to rollback and repeat statements such as `CREATE ... SELECT` on replicas.

```
CREATE OR REPLACE TABLE table_name (a int);
```

is basically the same as:

```
DROP TABLE IF EXISTS table_name;
CREATE TABLE table_name (a int);
```

with the following exceptions:

- If `table_name` was locked with LOCK TABLES it will continue to be locked after the statement.
- Temporary tables are only dropped if the `TEMPORARY` keyword was used. (With DROP TABLE, temporary tables are preferred to be dropped before normal tables).

Things to be Aware of With CREATE OR REPLACE

- The table is dropped first (if it existed), after that the `CREATE` is done. Because of this, if the `CREATE` fails, then the table will not exist anymore after the statement. If the table was used with `LOCK TABLES` it will be unlocked.
- One can't use `OR REPLACE` together with `IF EXISTS`.
- Slaves in replication will by default use `CREATE OR REPLACE` when replicating `CREATE` statements that don"t use `IF EXISTS`. This can be changed by setting the variable slave-ddl-exec-mode to `STRICT`.

CREATE TABLE IF NOT EXISTS

If the `IF NOT EXISTS` clause is used, then the table will only be created if a table with the same name does not already exist. If the table already exists, then a warning will be triggered by default.

CREATE TEMPORARY TABLE

Use the `TEMPORARY` keyword to create a temporary table that is only available to the current session. Temporary tables are dropped when the session ends. Temporary table names are specific to the session. They will not conflict with other temporary tables from other sessions even if they share the same name. They will shadow names of non-temporary tables or views, if they are identical. A temporary table can have the same name as a non-temporary table which is located in the same database. In that case, their name will reference the temporary table when used in SQL statements. You must have the CREATE TEMPORARY TABLES privilege on the database to create temporary tables. If no storage engine is specified, the default_tmp_storage_engine setting will determine the engine.

ROCKSDB temporary tables cannot be created by setting the default_tmp_storage_engine system variable, or using `CREATE TEMPORARY TABLE LIKE`. Before MariaDB 10.7, they could be specified, but would silently fail, and a MyISAM table would be created instead. From MariaDB 10.7 an error is returned. Explicitly creating a temporary table with `ENGINE=ROCKSDB` has never been permitted.

CREATE TABLE ... LIKE

Use the `LIKE` clause instead of a full table definition to create a table with the same definition as another table, including columns, indexes, and table options. Foreign key definitions, as well as any DATA DIRECTORY or INDEX DIRECTORY table options specified on the original table, will not be created.

CREATE TABLE ... SELECT

You can create a table containing data from other tables using the `CREATE ... SELECT` statement. Columns will be created in the table for

each field returned by the `SELECT` query.

You can also define some columns normally and add other columns from a `SELECT`. You can also create columns in the normal way and assign them some values using the query, this is done to force a certain type or other field characteristics. The columns that are not named in the query will be placed before the others. For example:

```
CREATE TABLE test (a INT NOT NULL, b CHAR(10)) ENGINE=MyISAM
    SELECT 5 AS b, c, d FROM another_table;
```

Remember that the query just returns data. If you want to use the same indexes, or the same columns attributes (`[NOT] NULL`, `DEFAULT`, `AUTO_INCREMENT`) in the new table, you need to specify them manually. Types and sizes are not automatically preserved if no data returned by the `SELECT` requires the full size, and `VARCHAR` could be converted into `CHAR`. The CAST() function can be used to forcee the new table to use certain types.

Aliases (`AS`) are taken into account, and they should always be used when you `SELECT` an expression (function, arithmetical operation, etc).

If an error occurs during the query, the table will not be created at all.

If the new table has a primary key or `UNIQUE` indexes, you can use the IGNORE or `REPLACE` keywords to handle duplicate key errors during the query. `IGNORE` means that the newer values must not be inserted an identical value exists in the index. `REPLACE` means that older values must be overwritten.

If the columns in the new table are more than the rows returned by the query, the columns populated by the query will be placed after other columns. Note that if the strict `SQL_MODE` is on, and the columns that are not names in the query do not have a `DEFAULT` value, an error will raise and no rows will be copied.

Concurrent inserts are not used during the execution of a `CREATE ... SELECT`.

If the table already exists, an error similar to the following will be returned:

```
ERROR 1050 (42S01): Table 't' already exists
```

If the `IF NOT EXISTS` clause is used and the table exists, a note will be produced instead of an error.

To insert rows from a query into an existing table, INSERT ... SELECT can be used.

Column Definitions

```
create_definition:
  { col_name column_definition | index_definition | period_definition | CHECK (expr) }

column_definition:
  data_type
    [NOT NULL | NULL] [DEFAULT default_value | (expression)]
    [ON UPDATE [NOW | CURRENT_TIMESTAMP] [(precision)]]
    [AUTO_INCREMENT] [ZEROFILL] [UNIQUE [KEY] | [PRIMARY] KEY]
    [INVISIBLE] [{WITH|WITHOUT} SYSTEM VERSIONING]
    [COMMENT 'string'] [REF_SYSTEM_ID = value]
    [reference_definition]
  | data_type [GENERATED ALWAYS]
  AS { { ROW {START|END} } | { (expression) [VIRTUAL | PERSISTENT | STORED] } }
      [UNIQUE [KEY]] [COMMENT 'string']

constraint_definition:
   CONSTRAINT [constraint_name] CHECK (expression)
```

Note: Until MariaDB 10.4, MariaDB accepts the shortcut format with a REFERENCES clause only in ALTER TABLE and CREATE TABLE statements, but that syntax does nothing. For example:

```
CREATE TABLE b(for_key INT REFERENCES a(not_key));
```

MariaDB simply parses it without returning any error or warning, for compatibility with other DBMS's. Before MariaDB 10.2.1 this was also true for `CHECK` constraints. However, only the syntax described below creates foreign keys.

From MariaDB 10.5, MariaDB will attempt to apply the constraint. See Foreign Keys examples.

Each definition either creates a column in the table or specifies and index or constraint on one or more columns. See Indexes below for details

on creating indexes.

Create a column by specifying a column name and a data type, optionally followed by column options. See Data Types for a full list of data types allowed in MariaDB.

NULL and NOT NULL

Use the `NULL` or `NOT NULL` options to specify that values in the column may or may not be `NULL`, respectively. By default, values may be `NULL`. See also NULL Values in MariaDB.

DEFAULT Column Option

MariaDB starting with 10.2.1

The `DEFAULT` clause was enhanced in MariaDB 10.2.1. Some enhancements include

- BLOB and TEXT columns now support `DEFAULT`.
- The `DEFAULT` clause can now be used with an expression or function.

Specify a default value using the `DEFAULT` clause. If you don't specify `DEFAULT` then the following rules apply:

- If the column is not defined with `NOT NULL`, `AUTO_INCREMENT` or `TIMESTAMP`, an explicit `DEFAULT NULL` will be added. Note that in MySQL and in MariaDB before 10.1.6, you may get an explicit `DEFAULT` for primary key parts, if not specified with NOT NULL.

The default value will be used if you INSERT a row without specifying a value for that column, or if you specify DEFAULT for that column. Before MariaDB 10.2.1 you couldn't usually provide an expression or function to evaluate at insertion time. You had to provide a constant default value instead. The one exception is that you may use CURRENT_TIMESTAMP as the default value for a TIMESTAMP column to use the current timestamp at insertion time.

CURRENT_TIMESTAMP may also be used as the default value for a DATETIME

From MariaDB 10.2.1 you can use most functions in `DEFAULT`. Expressions should have parentheses around them. If you use a non deterministic function in `DEFAULT` then all inserts to the table will be replicated in row mode. You can even refer to earlier columns in the `DEFAULT` expression (excluding `AUTO_INCREMENT` columns):

```
CREATE TABLE t1 (a int DEFAULT (1+1), b int DEFAULT (a+1));
CREATE TABLE t2 (a bigint primary key DEFAULT UUID_SHORT());
```

The `DEFAULT` clause cannot contain any stored functions or subqueries, and a column used in the clause must already have been defined earlier in the statement.

Since MariaDB 10.2.1, it is possible to assign BLOB or TEXT columns a `DEFAULT` value. In earlier versions, assigning a default to these columns was not possible.

MariaDB starting with 10.3.3

Starting from 10.3.3 you can also use DEFAULT (NEXT VALUE FOR sequence)

AUTO_INCREMENT Column Option

Use AUTO_INCREMENT to create a column whose value can can be set automatically from a simple counter. You can only use `AUTO_INCREMENT` on a column with an integer type. The column must be a key, and there can only be one `AUTO_INCREMENT` column in a table. If you insert a row without specifying a value for that column (or if you specify `0`, `NULL`, or DEFAULT as the value), the actual value will be taken from the counter, with each insertion incrementing the counter by one. You can still insert a value explicitly. If you insert a value that is greater than the current counter value, the counter is set based on the new value. An `AUTO_INCREMENT` column is implicitly `NOT NULL`. Use LAST_INSERT_ID to get the AUTO_INCREMENT value most recently used by an INSERT statement.

ZEROFILL Column Option

If the `ZEROFILL` column option is specified for a column using a numeric data type, then the column will be set to `UNSIGNED` and the spaces used by default to pad the field are replaced with zeros. `ZEROFILL` is ignored in expressions or as part of a UNION. `ZEROFILL` is a non-standard MySQL and MariaDB enhancement.

PRIMARY KEY Column Option

Use `PRIMARY KEY` to make a column a primary key. A primary key is a special type of a unique key. There can be at most one primary key per table, and it is implicitly `NOT NULL`.

Specifying a column as a unique key creates a unique index on that column. See the Index Definitions section below for more information.

UNIQUE KEY Column Option

Use `UNIQUE KEY` (or just `UNIQUE`) to specify that all values in the column must be distinct from each other. Unless the column is `NOT NULL`, there may be multiple rows with `NULL` in the column.

Specifying a column as a unique key creates a unique index on that column. See the Index Definitions section below for more information.

COMMENT Column Option

You can provide a comment for each column using the `COMMENT` clause. The maximum length is 1024 characters. Use the SHOW FULL COLUMNS statement to see column comments.

REF_SYSTEM_ID

`REF_SYSTEM_ID` can be used to specify Spatial Reference System IDs for spatial data type columns.

Generated Columns

A generated column is a column in a table that cannot explicitly be set to a specific value in a DML query. Instead, its value is automatically generated based on an expression. This expression might generate the value based on the values of other columns in the table, or it might generate the value by calling built-in functions or user-defined functions (UDFs).

There are two types of generated columns:

- `PERSISTENT` or `STORED`: This type's value is actually stored in the table.
- `VIRTUAL`: This type's value is not stored at all. Instead, the value is generated dynamically when the table is queried. This type is the default.

Generated columns are also sometimes called computed columns or virtual columns.

For a complete description about generated columns and their limitations, see Generated (Virtual and Persistent/Stored) Columns.

COMPRESSED

MariaDB starting with 10.3.3

Certain columns may be compressed. See Storage-Engine Independent Column Compression.

INVISIBLE

MariaDB starting with 10.3.3

Columns may be made invisible, and hidden in certain contexts. See Invisible Columns.

WITH SYSTEM VERSIONING Column Option

MariaDB starting with 10.3.4

Columns may be explicitly marked as included from system versioning. See System-versioned tables for details.

WITHOUT SYSTEM VERSIONING Column Option

MariaDB starting with 10.3.4

Columns may be explicitly marked as excluded from system versioning. See System-versioned tables for details.

Index Definitions

```
index_definition:
    {INDEX|KEY} [index_name] [index_type] (index_col_name,...) [index_option] ...
  | {FULLTEXT|SPATIAL} [INDEX|KEY] [index_name] (index_col_name,...) [index_option] ...
  | [CONSTRAINT [symbol]] PRIMARY KEY [index_type] (index_col_name,...) [index_option] ...
  | [CONSTRAINT [symbol]] UNIQUE [INDEX|KEY] [index_name] [index_type] (index_col_name,...) [index_option] ...
  | [CONSTRAINT [symbol]] FOREIGN KEY [index_name] (index_col_name,...) reference_definition

index_col_name:
    col_name [(length)] [ASC | DESC]

index_type:
    USING {BTREE | HASH | RTREE}

index_option:
    [ KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
  | CLUSTERING={YES| NO} ]
  [ IGNORED | NOT IGNORED ]

reference_definition:
    REFERENCES tbl_name (index_col_name,...)
      [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
      [ON DELETE reference_option]
      [ON UPDATE reference_option]

reference_option:
    RESTRICT | CASCADE | SET NULL | NO ACTION
```

`INDEX` and `KEY` are synonyms.

Index names are optional, if not specified an automatic name will be assigned. Index name are needed to drop indexes and appear in error messages when a constraint is violated.

Index Categories

Plain Indexes

Plain indexes are regular indexes that are not unique, and are not acting as a primary key or a foreign key. They are also not the "specialized" `FULLTEXT` or `SPATIAL` indexes.

See Getting Started with Indexes: Plain Indexes for more information.

PRIMARY KEY

For `PRIMARY KEY` indexes, you can specify a name for the index, but it is ignored, and the name of the index is always `PRIMARY`. From MariaDB 10.3.18 and MariaDB 10.4.8, a warning is explicitly issued if a name is specified. Before then, the name was silently ignored.

See Getting Started with Indexes: Primary Key for more information.

UNIQUE

The `UNIQUE` keyword means that the index will not accept duplicated values, except for NULLs. An error will raise if you try to insert duplicate values in a UNIQUE index.

For `UNIQUE` indexes, you can specify a name for the constraint, using the `CONSTRAINT` keyword. That name will be used in error messages.

See Getting Started with Indexes: Unique Index for more information.

FOREIGN KEY

For `FOREIGN KEY` indexes, a reference definition must be provided.

For `FOREIGN KEY` indexes, you can specify a name for the constraint, using the `CONSTRAINT` keyword. That name will be used in error messages.

First, you have to specify the name of the target (parent) table and a column or a column list which must be indexed and whose values must match to the foreign key's values. The `MATCH` clause is accepted to improve the compatibility with other DBMS's, but has no meaning in

MariaDB. The `ON DELETE` and `ON UPDATE` clauses specify what must be done when a `DELETE` (or a `REPLACE`) statements attempts to delete a referenced row from the parent table, and when an `UPDATE` statement attempts to modify the referenced foreign key columns in a parent table row, respectively. The following options are allowed:

- `RESTRICT`: The delete/update operation is not performed. The statement terminates with a 1451 error (SQLSTATE '2300').
- `NO ACTION`: Synonym for `RESTRICT`.
- `CASCADE`: The delete/update operation is performed in both tables.
- `SET NULL`: The update or delete goes ahead in the parent table, and the corresponding foreign key fields in the child table are set to `NULL`. (They must not be defined as `NOT NULL` for this to succeed).
- `SET DEFAULT`: This option is currently implemented only for the PBXT storage engine, which is disabled by default and no longer maintained. It sets the child table's foreign key fields to their `DEFAULT` values when the referenced parent table key entries are updated or deleted.

If either clause is omitted, the default behavior for the omitted clause is `RESTRICT`.

See Foreign Keys for more information.

FULLTEXT

Use the `FULLTEXT` keyword to create full-text indexes.

See Full-Text Indexes for more information.

SPATIAL

Use the `SPATIAL` keyword to create geometric indexes.

See SPATIAL INDEX for more information.

Index Options

KEY_BLOCK_SIZE Index Option

The `KEY_BLOCK_SIZE` index option is similar to the KEY_BLOCK_SIZE table option.

With the InnoDB storage engine, if you specify a non-zero value for the `KEY_BLOCK_SIZE` table option for the whole table, then the table will implicitly be created with the ROW_FORMAT table option set to `COMPRESSED`. However, this does not happen if you just set the `KEY_BLOCK_SIZE` index option for one or more indexes in the table. The InnoDB storage engine ignores the `KEY_BLOCK_SIZE` index option. However, the SHOW CREATE TABLE statement may still report it for the index.

For information about the `KEY_BLOCK_SIZE` index option, see the KEY_BLOCK_SIZE table option below.

Index Types

Each storage engine supports some or all index types. See Storage Engine Index Types for details on permitted index types for each storage engine.

Different index types are optimized for different kind of operations:

- `BTREE` is the default type, and normally is the best choice. It is supported by all storage engines. It can be used to compare a column's value with a value using the =, >, >=, <, <=, `BETWEEN`, and `LIKE` operators. `BTREE` can also be used to find `NULL` values. Searches against an index prefix are possible.
- `HASH` is only supported by the MEMORY storage engine. `HASH` indexes can only be used for =, <=, and >= comparisons. It can not be used for the `ORDER BY` clause. Searches against an index prefix are not possible.
- `RTREE` is the default for `SPATIAL` indexes, but if the storage engine does not support it `BTREE` can be used.

Index columns names are listed between parenthesis. After each column, a prefix length can be specified. If no length is specified, the whole column will be indexed. `ASC` and `DESC` can be specified for compatibility with are DBMS's, but have no meaning in MariaDB.

WITH PARSER Index Option

The `WITH PARSER` index option only applies to FULLTEXT indexes and contains the fulltext parser name. The fulltext parser must be an installed plugin.

COMMENT Index Option

A comment of up to 1024 characters is permitted with the `COMMENT` index option.

The `COMMENT` index option allows you to specify a comment with user-readable text describing what the index is for. This information is not used by the server itself.

CLUSTERING Index Option

The `CLUSTERING` index option is only valid for tables using the Tokudb storage engine.

IGNORED / NOT IGNORED

MariaDB starting with 10.6.0

From MariaDB 10.6.0, indexes can be specified to be ignored by the optimizer. See Ignored Indexes.

Periods

MariaDB starting with 10.3.4

```
period_definition:
    PERIOD FOR SYSTEM_TIME (start_column_name, end_column_name)
```

MariaDB supports a subset of the standard syntax for periods. At the moment it's only used for creating System-versioned tables. Both columns must be created, must be either of a `TIMESTAMP(6)` or `BIGINT UNSIGNED` type, and be generated as `ROW START` and `ROW END` accordingly. See System-versioned tables for details.

The table must also have the `WITH SYSTEM VERSIONING` clause.

Constraint Expressions

MariaDB starting with 10.2.1

MariaDB 10.2.1 introduced new ways to define a constraint.

Note: Before MariaDB 10.2.1, constraint expressions were accepted in the syntax but ignored.

MariaDB 10.2.1 introduced two ways to define a constraint:

- `CHECK(expression)` given as part of a column definition.
- `CONSTRAINT [constraint_name] CHECK (expression)`

Before a row is inserted or updated, all constraints are evaluated in the order they are defined. If any constraints fails, then the row will not be updated. One can use most deterministic functions in a constraint, including UDFs.

```
create table t1 (a int check(a>0) ,b int check (b> 0), constraint abc check (a>b));
```

If you use the second format and you don't give a name to the constraint, then the constraint will get a auto generated name. This is done so that you can later delete the constraint with ALTER TABLE DROP constraint_name.

One can disable all constraint expression checks by setting the variable `check_constraint_checks` to `OFF` . This is useful for example when loading a table that violates some constraints that you want to later find and fix in SQL.

See CONSTRAINT for more information.

Table Options

For each individual table you create (or alter), you can set some table options. The general syntax for setting options is:

<OPTION_NAME> = <option_value>, [<OPTION_NAME> = <option_value> ...]

The equal sign is optional.

Some options are supported by the server and can be used for all tables, no matter what storage engine they use; other options can be specified for all storage engines, but have a meaning only for some engines. Also, engines can extend `CREATE TABLE` with new options.

If the `IGNORE_BAD_TABLE_OPTIONS` SQL_MODE is enabled, wrong table options generate a warning; otherwise, they generate an error.

```
table_option:
    [STORAGE] ENGINE [=] engine_name
  | AUTO_INCREMENT [=] value
  | AVG_ROW_LENGTH [=] value
  | [DEFAULT] CHARACTER SET [=] charset_name
  | CHECKSUM [=] {0 | 1}
  | [DEFAULT] COLLATE [=] collation_name
  | COMMENT [=] 'string'
  | CONNECTION [=] 'connect_string'
  | DATA DIRECTORY [=] 'absolute path to directory'
  | DELAY_KEY_WRITE [=] {0 | 1}
  | ENCRYPTED [=] {YES | NO}
  | ENCRYPTION_KEY_ID [=] value
  | IETF_QUOTES [=] {YES | NO}
  | INDEX DIRECTORY [=] 'absolute path to directory'
  | INSERT_METHOD [=] { NO | FIRST | LAST }
  | KEY_BLOCK_SIZE [=] value
  | MAX_ROWS [=] value
  | MIN_ROWS [=] value
  | PACK_KEYS [=] {0 | 1 | DEFAULT}
  | PAGE_CHECKSUM [=] {0 | 1}
  | PAGE_COMPRESSED [=] {0 | 1}
  | PAGE_COMPRESSION_LEVEL [=] {0 .. 9}
  | PASSWORD [=] 'string'
  | ROW_FORMAT [=] {DEFAULT|DYNAMIC|FIXED|COMPRESSED|REDUNDANT|COMPACT|PAGE}
  | SEQUENCE [=] {0|1}
  | STATS_AUTO_RECALC [=] {DEFAULT|0|1}
  | STATS_PERSISTENT [=] {DEFAULT|0|1}
  | STATS_SAMPLE_PAGES [=] {DEFAULT|value}
  | TABLESPACE tablespace_name
  | TRANSACTIONAL [=]  {0 | 1}
  | UNION [=] (tbl_name[,tbl_name]...)
  | WITH SYSTEM VERSIONING
```

[STORAGE] ENGINE

`[STORAGE] ENGINE` specifies a storage engine for the table. If this option is not used, the default storage engine is used instead. That is, the default_storage_engine session option value if it is set, or the value specified for the `--default-storage-engine` mysqld startup option, or the default storage engine, InnoDB. If the specified storage engine is not installed and active, the default value will be used, unless the `NO_ENGINE_SUBSTITUTION` SQL MODE is set (default). This is only true for `CREATE TABLE` , not for `ALTER TABLE` . For a list of storage engines that are present in your server, issue a SHOW ENGINES.

AUTO_INCREMENT

`AUTO_INCREMENT` specifies the initial value for the `AUTO_INCREMENT` primary key. This works for MyISAM, Aria, InnoDB/XtraDB, MEMORY, and ARCHIVE tables. You can change this option with `ALTER TABLE` , but in that case the new value must be higher than the highest value which is present in the `AUTO_INCREMENT` column. If the storage engine does not support this option, you can insert (and then delete) a row having the wanted value - 1 in the `AUTO_INCREMENT` column.

AVG_ROW_LENGTH

`AVG_ROW_LENGTH` is the average rows size. It only applies to tables using MyISAM and Aria storage engines that have the ROW_FORMAT table option set to `FIXED` format.

MyISAM uses `MAX_ROWS` and `AVG_ROW_LENGTH` to decide the maximum size of a table (default: 256TB, or the maximum file size allowed by the system).

[DEFAULT] CHARACTER SET/CHARSET

`[DEFAULT] CHARACTER SET` (or `[DEFAULT] CHARSET`) is used to set a default character set for the table. This is the character set used for all columns where an explicit character set is not specified. If this option is omitted or `DEFAULT` is specified, database's default character set will be used. See Setting Character Sets and Collations for details on setting the character sets.

CHECKSUM/TABLE_CHECKSUM

`CHECKSUM` (or `TABLE_CHECKSUM`) can be set to 1 to maintain a live checksum for all table's rows. This makes write operations slower, but `CHECKSUM TABLE` will be very fast. This option is only supported for MyISAM and Aria tables.

[DEFAULT] COLLATE

`[DEFAULT] COLLATE` is used to set a default collation for the table. This is the collation used for all columns where an explicit character set is not specified. If this option is omitted or `DEFAULT` is specified, database's default option will be used. See Setting Character Sets and Collations for details on setting the collations

COMMENT

`COMMENT` is a comment for the table. The maximum length is 2048 characters. Also used to define table parameters when creating a Spider table.

CONNECTION

`CONNECTION` is used to specify a server name or a connection string for a Spider, CONNECT, Federated or FederatedX table.

DATA DIRECTORY/INDEX DIRECTORY

`DATA DIRECTORY` and `INDEX DIRECTORY` are supported for MyISAM and Aria, and DATA DIRECTORY is also supported by InnoDB if the innodb_file_per_table server system variable is enabled, but only in CREATE TABLE, not in ALTER TABLE. So, carefully choose a path for InnoDB tables at creation time, because it cannot be changed without dropping and re-creating the table. These options specify the paths for data files and index files, respectively. If these options are omitted, the database's directory will be used to store data files and index files. Note that these table options do not work for partitioned tables (use the partition options instead), or if the server has been invoked with the `--skip-symbolic-links` startup option. To avoid the overwriting of old files with the same name that could be present in the directories, you can use the `--keep_files_on_create` option (an error will be issued if files already exist). These options are ignored if the `NO_DIR_IN_CREATE` SQL_MODE is enabled (useful for replication slaves). Also note that symbolic links cannot be used for InnoDB tables.

`DATA DIRECTORY` works by creating symlinks from where the table would normally have been (inside the datadir) to where the option specifies. For security reasons, to avoid bypassing the privilege system, the server does not permit symlinks inside the datadir. Therefore, `DATA DIRECTORY` cannot be used to specify a location inside the datadir. An attempt to do so will result in an error `1210 (HY000) Incorrect arguments to DATA DIRECTORY`.

DELAY_KEY_WRITE

`DELAY_KEY_WRITE` is supported by MyISAM and Aria, and can be set to 1 to speed up write operations. In that case, when data are modified, the indexes are not updated until the table is closed. Writing the changes to the index file altogether can be much faster. However, note that this option is applied only if the `delay_key_write` server variable is set to 'ON'. If it is 'OFF' the delayed index writes are always disabled, and if it is 'ALL' the delayed index writes are always used, disregarding the value of `DELAY_KEY_WRITE`.

ENCRYPTED

The `ENCRYPTED` table option can be used to manually set the encryption status of an InnoDB table. See InnoDB Encryption for more information.

Aria does not support the `ENCRYPTED` table option. See MDEV-18049.

See Data-at-Rest Encryption for more information.

ENCRYPTION_KEY_ID

The `ENCRYPTION_KEY_ID` table option can be used to manually set the encryption key of an InnoDB table. See InnoDB Encryption for more information.

Aria does not support the `ENCRYPTION_KEY_ID` table option. See MDEV-18049.

See Data-at-Rest Encryption for more information.

IETF_QUOTES

For the CSV storage engine, the `IETF_QUOTES` option, when set to `YES`, enables IETF-compatible parsing of embedded quote and comma characters. Enabling this option for a table improves compatibility with other tools that use CSV, but is not compatible with MySQL CSV tables, or MariaDB CSV tables created without this option. Disabled by default.

INSERT_METHOD

`INSERT_METHOD` is only used with MERGE tables. This option determines in which underlying table the new rows should be inserted. If you set it to 'NO' (which is the default) no new rows can be added to the table (but you will still be able to perform `INSERT` s directly against the underlying tables). `FIRST` means that the rows are inserted into the first table, and `LAST` means that thet are inserted into the last table.

KEY_BLOCK_SIZE

`KEY_BLOCK_SIZE` is used to determine the size of key blocks, in bytes or kilobytes. However, this value is just a hint, and the storage engine could modify or ignore it. If `KEY_BLOCK_SIZE` is set to 0, the storage engine's default value will be used.

With the InnoDB storage engine, if you specify a non-zero value for the `KEY_BLOCK_SIZE` table option for the whole table, then the table will implicitly be created with the ROW_FORMAT table option set to `COMPRESSED` .

MIN_ROWS/MAX_ROWS

`MIN_ROWS` and `MAX_ROWS` let the storage engine know how many rows you are planning to store as a minimum and as a maximum. These values will not be used as real limits, but they help the storage engine to optimize the table. `MIN_ROWS` is only used by MEMORY storage engine to decide the minimum memory that is always allocated. `MAX_ROWS` is used to decide the minimum size for indexes.

PACK_KEYS

`PACK_KEYS` can be used to determine whether the indexes will be compressed. Set it to 1 to compress all keys. With a value of 0, compression will not be used. With the `DEFAULT` value, only long strings will be compressed. Uncompressed keys are faster.

PAGE_CHECKSUM

`PAGE_CHECKSUM` is only applicable to Aria tables, and determines whether indexes and data should use page checksums for extra safety.

PAGE_COMPRESSED

`PAGE_COMPRESSED` is used to enable InnoDB page compression for InnoDB tables.

PAGE_COMPRESSION_LEVEL

`PAGE_COMPRESSION_LEVEL` is used to set the compression level for InnoDB page compression for InnoDB tables. The table must also have the PAGE_COMPRESSED table option set to `1` .

Valid values for `PAGE_COMPRESSION_LEVEL` are 1 (the best speed) through 9 (the best compression), .

PASSWORD

`PASSWORD` is unused.

RAID_TYPE

`RAID_TYPE` is an obsolete option, as the raid support has been disabled since MySQL 5.0.

ROW_FORMAT

The `ROW_FORMAT` table option specifies the row format for the data file. Possible values are engine-dependent.

Supported MyISAM Row Formats

For MyISAM, the supported row formats are:

- `FIXED`
- `DYNAMIC`
- `COMPRESSED`

The `COMPRESSED` row format can only be set by the myisampack command line tool.

See MyISAM Storage Formats for more information.

Supported Aria Row Formats

For Aria, the supported row formats are:

- `PAGE`
- `FIXED`
- `DYNAMIC`.

See Aria Storage Formats for more information.

Supported InnoDB Row Formats

For InnoDB, the supported row formats are:

- `COMPACT`
- `REDUNDANT`
- `COMPRESSED`
- `DYNAMIC`.

If the `ROW_FORMAT` table option is set to `FIXED` for an InnoDB table, then the server will either return an error or a warning depending on the value of the innodb_strict_mode system variable. If the innodb_strict_mode system variable is set to `OFF`, then a warning is issued, and MariaDB will create the table using the default row format for the specific MariaDB server version. If the innodb_strict_mode system variable is set to `ON`, then an error will be raised.

See InnoDB Storage Formats for more information.

Other Storage Engines and ROW_FORMAT

Other storage engines do not support the `ROW_FORMAT` table option.

SEQUENCE

MariaDB starting with 10.3

If the table is a sequence, then it will have the `SEQUENCE` set to `1`.

STATS_AUTO_RECALC

`STATS_AUTO_RECALC` indicates whether to automatically recalculate persistent statistics (see `STATS_PERSISTENT`, below) for an InnoDB table. If set to `1`, statistics will be recalculated when more than 10% of the data has changed. When set to `0`, stats will be recalculated only when an ANALYZE TABLE is run. If set to `DEFAULT`, or left out, the value set by the innodb_stats_auto_recalc system variable applies. See InnoDB Persistent Statistics.

STATS_PERSISTENT

`STATS_PERSISTENT` indicates whether the InnoDB statistics created by ANALYZE TABLE will remain on disk or not. It can be set to `1` (on disk), `0` (not on disk, the pre-MariaDB 10 behavior), or `DEFAULT` (the same as leaving out the option), in which case the value set by the innodb_stats_persistent system variable will apply. Persistent statistics stored on disk allow the statistics to survive server restarts, and provide better query plan stability. See InnoDB Persistent Statistics.

STATS_SAMPLE_PAGES

`STATS_SAMPLE_PAGES` indicates how many pages are used to sample index statistics. If 0 or DEFAULT, the default value, the innodb_stats_sample_pages value is used. See InnoDB Persistent Statistics.

TRANSACTIONAL

`TRANSACTIONAL` is only applicable for Aria tables. In future Aria tables created with this option will be fully transactional, but currently this provides a form of crash protection. See Aria Storage Engine for more details.

UNION

`UNION` must be specified when you create a MERGE table. This option contains a comma-separated list of MyISAM tables which are accessed by the new table. The list is enclosed between parenthesis. Example: `UNION = (t1,t2)`

WITH SYSTEM VERSIONING

`WITH SYSTEM VERSIONING` is used for creating System-versioned tables.

Partitions

```
partition_options:
    PARTITION BY
        { [LINEAR] HASH(expr)
        | [LINEAR] KEY(column_list)
        | RANGE(expr)
        | LIST(expr)
        | SYSTEM_TIME [INTERVAL time_quantity time_unit] [LIMIT num] }
    [PARTITIONS num]
    [SUBPARTITION BY
        { [LINEAR] HASH(expr)
        | [LINEAR] KEY(column_list) }
      [SUBPARTITIONS num]
    ]
    [(partition_definition [, partition_definition] ...)]

partition_definition:
    PARTITION partition_name
        [VALUES {LESS THAN {(expr) | MAXVALUE} | IN (value_list)}]
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'comment_text' ]
        [DATA DIRECTORY [=] 'data_dir']
        [INDEX DIRECTORY [=] 'index_dir']
        [MAX_ROWS [=] max_number_of_rows]
        [MIN_ROWS [=] min_number_of_rows]
        [TABLESPACE [=] tablespace_name]
        [NODEGROUP [=] node_group_id]
        [(subpartition_definition [, subpartition_definition] ...)]

subpartition_definition:
    SUBPARTITION logical_name
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'comment_text' ]
        [DATA DIRECTORY [=] 'data_dir']
        [INDEX DIRECTORY [=] 'index_dir']
        [MAX_ROWS [=] max_number_of_rows]
        [MIN_ROWS [=] min_number_of_rows]
        [TABLESPACE [=] tablespace_name]
        [NODEGROUP [=] node_group_id]
```

If the `PARTITION BY` clause is used, the table will be partitioned. A partition method must be explicitly indicated for partitions and subpartitions. Partition methods are:

- `[LINEAR] HASH` creates a hash key which will be used to read and write rows. The partition function can be any valid SQL expression which returns an `INTEGER` number. Thus, it is possible to use the `HASH` method on an integer column, or on functions which accept integer columns as an argument. However, `VALUES LESS THAN` and `VALUES IN` clauses can not be used with `HASH` . An example:

```
CREATE TABLE t1 (a INT, b CHAR(5), c DATETIME)
    PARTITION BY HASH ( YEAR(c) );
```

`[LINEAR] HASH` can be used for subpartitions, too.

- `[LINEAR] KEY` is similar to `HASH` , but the index has an even distribution of data. Also, the expression can only be a column or a list of columns. `VALUES LESS THAN` and `VALUES IN` clauses can not be used with `KEY` .
- RANGE partitions the rows using on a range of values, using the `VALUES LESS THAN` operator. `VALUES IN` is not allowed with `RANGE` . The partition function can be any valid SQL expression which returns a single value.
- LIST assigns partitions based on a table's column with a restricted set of possible values. It is similar to `RANGE` , but `VALUES IN` must be used for at least 1 columns, and `VALUES LESS THAN` is disallowed.
- `SYSTEM_TIME` partitioning is used for System-versioned tables to store historical data separately from current data.

Only `HASH` and `KEY` can be used for subpartitions, and they can be `[LINEAR]`.

It is possible to define up to 1024 partitions and subpartitions.

The number of defined partitions can be optionally specified as `PARTITION count`. This can be done to avoid specifying all partitions individually. But you can also declare each individual partition and, additionally, specify a `PARTITIONS count` clause; in the case, the number of `PARTITION`s must equal count.

Also see Partitioning Types Overview.

Sequences

MariaDB starting with 10.3

`CREATE TABLE` can also be used to create a SEQUENCE. See CREATE SEQUENCE and Sequence Overview.

Atomic DDL

MariaDB starting with 10.6.1

MariaDB 10.6.1 supports Atomic DDL. `CREATE TABLE` is atomic, except for `CREATE OR REPLACE`, which is only crash safe.

Examples

```
create table if not exists test (
a bigint auto_increment primary key,
name varchar(128) charset utf8,
key name (name(32))
) engine=InnoDB default charset latin1;
```

This example shows a couple of things:

- Usage of `IF NOT EXISTS`; If the table already existed, it will not be created. There will not be any error for the client, just a warning.
- How to create a `PRIMARY KEY` that is automatically generated.
- How to specify a table-specific character set and another for a column.
- How to create an index (`name`) that is only partly indexed (to save space).

The following clauses will work from MariaDB 10.2.1 only.

```
CREATE TABLE t1(
  a int DEFAULT (1+1),
  b int DEFAULT (a+1),
  expires DATETIME DEFAULT(NOW() + INTERVAL 1 YEAR),
  x BLOB DEFAULT USER()
);
```

See Also

- Identifier Names
- ALTER TABLE
- DROP TABLE
- Character Sets and Collations
- SHOW CREATE TABLE
- Storage engines can add their own attributes for columns, indexes and tables.
- Variable slave-ddl-exec-mode.

1.1.2.1.8 DELETE

Contents

Syntax

Single-table syntax:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
  FROM tbl_name [PARTITION (partition_list)]
  [FOR PORTION OF period FROM expr1 TO expr2]
  [WHERE where_condition]
  [ORDER BY ...]
  [LIMIT row_count]
  [RETURNING select_expr
    [, select_expr ...]]
```

Multiple-table syntax:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
    tbl_name[.*] [, tbl_name[.*]] ...
    FROM table_references
    [WHERE where_condition]
```

Or:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
    FROM tbl_name[.*] [, tbl_name[.*]] ...
    USING table_references
    [WHERE where_condition]
```

Trimming history:

```
DELETE HISTORY
  FROM tbl_name [PARTITION (partition_list)]
  [BEFORE SYSTEM_TIME [TIMESTAMP|TRANSACTION] expression]
```

Description

Option	Description
LOW_PRIORITY	Wait until all SELECT's are done before starting the statement. Used with storage engines that uses table locking (MyISAM, Aria etc). See HIGH_PRIORITY and LOW_PRIORITY clauses for details.
QUICK	Signal the storage engine that it should expect that a lot of rows are deleted. The storage engine engine can do things to speed up the DELETE like ignoring merging of data blocks until all rows are deleted from the block (instead of when a block is half full). This speeds up things at the expanse of lost space in data blocks. At least MyISAM and Aria support this feature.
IGNORE	Don't stop the query even if a not-critical error occurs (like data overflow). See How IGNORE works for a full description.

For the single-table syntax, the `DELETE` statement deletes rows from tbl_name and returns a count of the number of deleted rows. This count can be obtained by calling the ROW_COUNT() function. The `WHERE` clause, if given, specifies the conditions that identify which rows to delete. With no `WHERE` clause, all rows are deleted. If the ORDER BY clause is specified, the rows are deleted in the order that is specified.

The LIMIT clause places a limit on the number of rows that can be deleted.

For the multiple-table syntax, `DELETE` deletes from each `tbl_name` the rows that satisfy the conditions. In this case, ORDER BY and LIMIT> cannot be used. A `DELETE` can also reference tables which are located in different databases; see Identifier Qualifiers for the syntax.

`where_condition` is an expression that evaluates to true for each row to be deleted. It is specified as described in SELECT.

Currently, you cannot delete from a table and select from the same table in a subquery.

You need the `DELETE` privilege on a table to delete rows from it. You need only the `SELECT` privilege for any columns that are only read, such as those named in the `WHERE` clause. See GRANT.

As stated, a `DELETE` statement with no `WHERE` clause deletes all rows. A faster way to do this, when you do not need to know the number of deleted rows, is to use `TRUNCATE TABLE`. However, within a transaction or if you have a lock on the table, `TRUNCATE TABLE` cannot be used whereas `DELETE` can. See `TRUNCATE TABLE`, and `LOCK`.

PARTITION

See Partition Pruning and Selection for details.

FOR PORTION OF

MariaDB starting with 10.4.3

See Application Time Periods - Deletion by Portion.

RETURNING

It is possible to return a resultset of the deleted rows for a single table to the client by using the syntax `DELETE ... RETURNING select_expr [, select_expr2 ...]]`

Any of SQL expression that can be calculated from a single row fields is allowed. Subqueries are allowed. The AS keyword is allowed, so it is possible to use aliases.

The use of aggregate functions is not allowed. RETURNING cannot be used in multi-table DELETEs.

MariaDB starting with 10.3.1

Same Source and Target Table

Until MariaDB 10.3.1, deleting from a table with the same source and target was not possible. From MariaDB 10.3.1, this is now possible. For example:

```
DELETE FROM t1 WHERE c1 IN (SELECT b.c1 FROM t1 b WHERE b.c2=0);
```

MariaDB starting with 10.3.4

DELETE HISTORY

One can use `DELETE HISTORY` to delete historical information from System-versioned tables.

Examples

How to use the ORDER BY and LIMIT clauses:

```
DELETE FROM page_hit ORDER BY timestamp LIMIT 1000000;
```

How to use the RETURNING clause:

```
DELETE FROM t RETURNING f1;
+------+
| f1   |
+------+
|    5 |
|   50 |
|  500 |
+------+
```

The following statement joins two tables: one is only used to satisfy a WHERE condition, but no row is deleted from it; rows from the other table are deleted, instead.

```
DELETE post FROM blog INNER JOIN post WHERE blog.id = post.blog_id;
```

Deleting from the Same Source and Target

```
CREATE TABLE t1 (c1 INT, c2 INT);
DELETE FROM t1 WHERE c1 IN (SELECT b.c1 FROM t1 b WHERE b.c2=0);
```

Until MariaDB 10.3.1, this returned:

```
ERROR 1093 (HY000): Table 't1' is specified twice, both as a target for 'DELETE'
  and as a separate source for
```

From MariaDB 10.3.1:

```
Query OK, 0 rows affected (0.00 sec)
```

See Also

- How IGNORE works
- SELECT
- ORDER BY
- LIMIT
- REPLACE ... RETURNING
- INSERT ... RETURNING
- Returning clause (video)

1.1.2.1.9 DROP TABLE

Syntax

```
DROP [TEMPORARY] TABLE [IF EXISTS] [/*COMMENT TO SAVE*/]
    tbl_name [, tbl_name] ...
    [WAIT n|NOWAIT]
    [RESTRICT | CASCADE]
```

Contents

Description

`DROP TABLE` removes one or more tables. You must have the `DROP` privilege for each table. All table data and the table definition are removed, as well as triggers associated to the table, so be careful with this statement! If any of the tables named in the argument list do not exist, MariaDB returns an error indicating by name which non-existing tables it was unable to drop, but it also drops all of the tables in the list that do exist.

Important: When a table is dropped, user privileges on the table are not automatically dropped. See GRANT.

If another thread is using the table in an explicit transaction or an autocommit transaction, then the thread acquires a metadata lock (MDL) on the table. The `DROP TABLE` statement will wait in the "Waiting for table metadata lock" thread state until the MDL is released. MDLs are released in the following cases:

- If an MDL is acquired in an explicit transaction, then the MDL will be released when the transaction ends.
- If an MDL is acquired in an autocommit transaction, then the MDL will be released when the statement ends.
- Transactional and non-transactional tables are handled the same.

Note that for a partitioned table, `DROP TABLE` permanently removes the table definition, all of its partitions, and all of the data which was stored in those partitions. It also removes the partitioning definition (.par) file associated with the dropped table.

For each referenced table, `DROP TABLE` drops a temporary table with that name, if it exists. If it does not exist, and the `TEMPORARY` keyword is not used, it drops a non-temporary table with the same name, if it exists. The `TEMPORARY` keyword ensures that a non-temporary table will not accidentally be dropped.

Use `IF EXISTS` to prevent an error from occurring for tables that do not exist. A `NOTE` is generated for each non-existent table when using `IF EXISTS`. See SHOW WARNINGS.

If a foreign key references this table, the table cannot be dropped. In this case, it is necessary to drop the foreign key first.

`RESTRICT` and `CASCADE` are allowed to make porting from other database systems easier. In MariaDB, they do nothing.

The comment before the table names (`/*COMMENT TO SAVE*/`) is stored in the binary log. That feature can be used by replication tools to send their internal messages.

It is possible to specify table names as `db_name.tab_name`. This is useful to delete tables from multiple databases with one statement. See Identifier Qualifiers for details.

The DROP privilege is required to use `DROP TABLE` on non-temporary tables. For temporary tables, no privilege is required, because such tables are only visible for the current session.

Note: `DROP TABLE` automatically commits the current active transaction, unless you use the `TEMPORARY` keyword.

MariaDB starting with 10.5.4

From MariaDB 10.5.4, `DROP TABLE` reliably deletes table remnants inside a storage engine even if the `.frm` file is missing. Before then, a missing `.frm` file would result in the statement failing.

MariaDB starting with 10.3.1

WAIT/NOWAIT

Set the lock wait timeout. See WAIT and NOWAIT.

DROP TABLE in replication

`DROP TABLE` has the following characteristics in replication:

- `DROP TABLE IF EXISTS` are always logged.
- `DROP TABLE` without `IF EXISTS` for tables that don't exist are not written to the binary log.
- Dropping of `TEMPORARY` tables are prefixed in the log with `TEMPORARY`. These drops are only logged when running statement or mixed mode replication.
- One `DROP TABLE` statement can be logged with up to 3 different `DROP` statements:
  - `DROP TEMPORARY TABLE list_of_non_transactional_temporary_tables`
  - `DROP TEMPORARY TABLE list_of_transactional_temporary_tables`
  - `DROP TABLE list_of_normal_tables`

`DROP TABLE` on the primary is treated on the replica as `DROP TABLE IF EXISTS`. You can change that by setting slave-ddl-exec-mode to `STRICT`.

Dropping an Internal #sql-... Table

From MariaDB 10.6, DROP TABLE is atomic and the following does not apply. Until MariaDB 10.5, if the mariadbd/mysqld process is killed during an ALTER TABLE you may find a table named #sql-... in your data directory. In MariaDB 10.3, InnoDB tables with this prefix will be deleted automatically during startup. From MariaDB 10.4, these temporary tables will always be deleted automatically.

If you want to delete one of these tables explicitly you can do so by using the following syntax:

```
DROP TABLE `#mysql50##sql-...`;
```

When running an `ALTER TABLE…ALGORITHM=INPLACE` that rebuilds the table, InnoDB will create an internal `#sql-ib` table. Until MariaDB 10.3.2, for these tables, the `.frm` file will be called something else. In order to drop such a table after a server crash, you must rename the `#sql*.frm` file to match the `#sql-ib*.ibd` file.

From MariaDB 10.3.3, the same name as the .frm file is used for the intermediate copy of the table. The #sql-ib names are used by TRUNCATE and delayed DROP.

From MariaDB 10.2.19 and MariaDB 10.3.10, the #sql-ib tables will be deleted automatically.

Dropping All Tables in a Database

The best way to drop all tables in a database is by executing `DROP DATABASE`, which will drop the database itself, and all tables in it.

However, if you want to drop all tables in the database, but you also want to keep the database itself and any other non-table objects in it, then you would need to execute `DROP TABLE` to drop each individual table. You can construct these `DROP TABLE` commands by querying the `TABLES` table in the `information_schema` database. For example:

```
SELECT CONCAT('DROP TABLE IF EXISTS `', TABLE_SCHEMA, '`.`', TABLE_NAME, '`;')
FROM information_schema.TABLES
WHERE TABLE_SCHEMA = 'mydb';
```

Atomic DROP TABLE

MariaDB starting with 10.6.1

From MariaDB 10.6, `DROP TABLE` for a single table is atomic (MDEV-25180) for most engines, including InnoDB, MyRocks, MyISAM and Aria.

This means that if there is a crash (server down or power outage) during `DROP TABLE`, all tables that have been processed so far will be completely dropped, including related trigger files and status entries, and the binary log will include a `DROP TABLE` statement for the dropped tables. Tables for which the drop had not started will be left intact.

In older MariaDB versions, there was a small chance that, during a server crash happening in the middle of `DROP TABLE`, some storage engines that were using multiple storage files, like MyISAM, could have only a part of its internal files dropped.

In MariaDB 10.5, `DROP TABLE` was extended to be able to delete a table that was only partly dropped (MDEV-11412) as explained above. Atomic `DROP TABLE` is the final piece to make `DROP TABLE` fully reliable.

Dropping multiple tables is crash-safe.

See Atomic DDL for more information.

Examples

```
DROP TABLE Employees, Customers;
```

Notes

Beware that `DROP TABLE` can drop both tables and sequences. This is mainly done to allow old tools like mysqldump to work with sequences.

See Also

- CREATE TABLE
- ALTER TABLE
- SHOW CREATE TABLE
- DROP SEQUENCE
- Variable slave-ddl-exec-mode.

1.1.2.1.10 Installing System Tables (mysql_install_db)

`mysql_install_db` initializes the MariaDB data directory and creates the system tables in the `mysql` database, if they do not exist. MariaDB uses these tables to manage privileges, roles, and plugins. It also uses them to provide the data for the `help` command in the `mysql` client.

`mysql_install_db` works by starting MariaDB Server's `mysqld` process in `--bootstrap` mode and sending commands to create the system tables and their content.

There is a version specifically for Windows, `mysql_install_db.exe` .

To invoke `mysql_install_db` , use the following syntax:

```
mysql_install_db --user=mysql
```

For the options supported by `mysql_install_db` , see mysql_install_db: Options.

For the option groups read by `mysql_install_db` , see mysql_install_db: Option Groups.

See mysql_install_db: Installing System Tables for information on the installation process.

See mysql_install_db: Troubleshooting Issues for information on how to troubleshoot the installation process.

See also:

- mysql_install_db
- The Windows version of `mysql_install_db` : `mysql_install_db.exe`

1.1.2.1.11 mysqlcheck

MariaDB starting with 10.4.6

From MariaDB 10.4.6, `mariadb-check` is a symlink to `mysqlcheck` .

MariaDB starting with 10.5.2

From MariaDB 10.5.2, `mariadb-check` is the name of the tool, with `mysqlcheck` a symlink .

Contents

`mysqlcheck` is a maintenance tool that allows you to check, repair, analyze and optimize multiple tables from the command line.

It is essentially a commandline interface to the CHECK TABLE, REPAIR TABLE, ANALYZE TABLE and OPTIMIZE TABLE commands, and so, unlike myisamchk and aria_chk, requires the server to be running.

This tool does not work with partitioned tables.

Using mysqlcheck

```
./client/mysqlcheck [OPTIONS] database [tables]
```

OR

```
./client/mysqlcheck [OPTIONS] --databases DB1 [DB2 DB3...]
```

OR

```
./client/mysqlcheck [OPTIONS] --all-databases
```

`mysqlcheck` can be used to CHECK (-c, -m, -C), REPAIR (-r), ANALYZE (-a), or OPTIMIZE (-o) tables. Some of the options (like -e or -q) can be used at the same time. Not all options are supported by all storage engines.

The -c, -r, -a and -o options are exclusive to each other.

The option `--check` will be used by default, if no other options were specified. You can change the default behavior by making a symbolic link to the binary, or copying it somewhere with another name, the alternatives are:

`mysqlrepair`	The default option will be `-r` (`--repair`)
`mysqlanalyze`	The default option will be `-a` (`--analyze`)
`mysqloptimize`	The default option will be `-o` (`--optimize`)

Options

`mysqlcheck` supports the following options:

Option	Description
`-A` , `--all-databases`	Check all the databases. This is the same as `--databases` with all databases selected.
`-1` , `--all-in-1`	Instead of issuing one query for each table, use one query per database, naming all tables in the database in a comma-separated list.
`-a` , `--analyze`	Analyze given tables.
`--auto-repair`	If a checked table is corrupted, automatically fix it. Repairing will be done after all tables have been checked.
`--character-sets-dir=name`	Directory where character set files are installed.
`-c` , `--check`	Check table for errors.
`-C` , `--check-only-changed`	Check only tables that have changed since last check or haven't been closed properly.
`-g` , `--check-upgrade`	Check tables for version-dependent changes. May be used with `--auto-repair` to correct tables requiring version-dependent updates. Automatically enables the `--fix-db-names` and `--fix-table-names` options. Used when upgrading
`--compress`	Compress all information sent between the client and server if both support compression.
`-B` , `--databases`	Check several databases. Note that normally *mysqlcheck* treats the first argument as a database name, and following arguments as table names. With this option, no tables are given, and all name arguments are regarded as database names.
`-#` , `--debug[=name]`	Output debug log. Often this is 'd:t:o,filename'.
`--debug-check`	Check memory and open file usage at exit.
`--debug-info`	Print some debug info at exit.
`--default-auth=plugin`	Default authentication client-side plugin to use.
`--default-character-set=name`	Set the default character set.

`-e, --extended`	If you are using this option with `--check`, it will ensure that the table is 100 percent consistent, but will take a long time. If you are using this option with `--repair`, it will force using the old, slow, repair with keycache method, instead of the much faster repair by sorting.
`-F, --fast`	Check only tables that haven't been closed properly.
`--fix-db-names`	Convert database names to the format used since MySQL 5.1. Only database names that contain special characters are affected. Used when upgrading from an old MySQL version.
`--fix-table-names`	Convert table names (including views) to the format used since MySQL 5.1. Only table names that contain special characters are affected. Used when upgrading from an old MySQL version.
`--flush`	Flush each table after check. This is useful if you don't want to have the checked tables take up space in the caches after the check.
`-f, --force`	Continue even if we get an SQL error.
`-?, --help`	Display this help message and exit.
`-h name, --host=name`	Connect to the given host.
`-m, --medium-check`	Faster than extended-check, but only finds 99.99 percent of all errors. Should be good enough for most cases.
`-o, --optimize`	Optimize tables.
`-p, --password[=name]`	Password to use when connecting to the server. If you use the short option form (`-p`), you cannot have a space between the option and the password. If you omit the password value following the `--password` or `-p` option on the command line, mysqlcheck prompts for one. Specifying a password on the command line should be considered insecure. You can use an option file to avoid giving the password on the command line.
`-Z, --persistent`	When using ANALYZE TABLE (`--analyze`), uses the PERSISTENT FOR ALL option, which forces Engine-independent Statistics for this table to be updated. Added in MariaDB 10.1.10
`-W, --pipe`	On Windows, connect to the server via a named pipe. This option applies only if the server supports named-pipe connections.
`--plugin-dir`	Directory for client-side plugins.
`-P num, --port=num`	Port number to use for connection or 0 for default to, in order of preference, my.cnf, $MYSQL_TCP_PORT, /etc/services, built-in default (3306).
`--process-tables`	Perform the requested operation (check, repair, analyze, optimize) on tables. Enabled by default. Use `--skip-process-tables` to disable.
`--process-views[=val]`	Perform the requested operation (only CHECK VIEW or REPAIR VIEW). Possible values are NO, YES (correct the checksum, if necessary, add the mariadb-version field), UPGRADE_FROM_MYSQL (same as YES and toggle the algorithm MERGE<->TEMPTABLE.
`--protocol=name`	The connection protocol (tcp, socket, pipe, memory) to use for connecting to the server. Useful when other connection parameters would cause a protocol to be used other than the one you want.
`-q, --quick`	If you are using this option with CHECK TABLE, it prevents the check from scanning the rows to check for wrong links. This is the fastest check. If you are using this option with REPAIR TABLE, it will try to repair only the index tree. This is the fastest repair method for a table.
`-r, --repair`	Can fix almost anything except unique keys that aren't unique.
`--shared-memory-base-name`	Shared-memory name to use for Windows connections using shared memory to a local server (started with the `--shared-memory` option). Case-sensitive.
`-s, --silent`	Print only error messages.
`--skip-database`	Don't process the database (case-sensitive) specified as argument.
`-S name, --socket=name`	For connections to localhost, the Unix socket file to use, or, on Windows, the name of the named pipe to use.
`--ssl`	Enables TLS. TLS is also enabled even without setting this option when certain other TLS options are set. Starting with MariaDB 10.2, the `--ssl` option will not enable verifying the server certificate by default. In order to verify the server certificate, the user must specify the `--ssl-verify-server-cert` option.

`--ssl-ca=name`	Defines a path to a PEM file that should contain one or more X509 certificates for trusted Certificate Authorities (CAs) to use for TLS. This option requires that you use the absolute path, not a relative path. See Secure Connections Overview: Certificate Authorities (CAs) for more information. This option implies the `--ssl` option.
`--ssl-capath=name`	Defines a path to a directory that contains one or more PEM files that should each contain one X509 certificate for a trusted Certificate Authority (CA) to use for TLS. This option requires that you use the absolute path, not a relative path. The directory specified by this option needs to be run through the `openssl rehash` command. See Secure Connections Overview: Certificate Authorities (CAs) for more information. This option is only supported if the client was built with OpenSSL or yaSSL. If the client was built with GnuTLS or Schannel, then this option is not supported. See TLS and Cryptography Libraries Used by MariaDB for more information about which libraries are used on which platforms. This option implies the `--ssl` option.
`--ssl-cert=name`	Defines a path to the X509 certificate file to use for TLS. This option requires that you use the absolute path, not a relative path. This option implies the `--ssl` option.
`--ssl-cipher=name`	List of permitted ciphers or cipher suites to use for TLS. This option implies the `--ssl` option.
`--ssl-crl=name`	Defines a path to a PEM file that should contain one or more revoked X509 certificates to use for TLS. This option requires that you use the absolute path, not a relative path. See Secure Connections Overview: Certificate Revocation Lists (CRLs) for more information. This option is only supported if the client was built with OpenSSL or Schannel. If the client was built with yaSSL or GnuTLS, then this option is not supported. See TLS and Cryptography Libraries Used by MariaDB for more information about which libraries are used on which platforms.
`--ssl-crlpath=name`	Defines a path to a directory that contains one or more PEM files that should each contain one revoked X509 certificate to use for TLS. This option requires that you use the absolute path, not a relative path. The directory specified by this option needs to be run through the `openssl rehash` command. See Secure Connections Overview: Certificate Revocation Lists (CRLs) for more information. This option is only supported if the client was built with OpenSSL. If the client was built with yaSSL, GnuTLS, or Schannel, then this option is not supported. See TLS and Cryptography Libraries Used by MariaDB for more information about which libraries are used on which platforms.
`--ssl-key=name`	Defines a path to a private key file to use for TLS. This option requires that you use the absolute path, not a relative path. This option implies the `--ssl` option.
`--ssl-verify-server-cert`	Enables server certificate verification. This option is disabled by default.
`--tables`	Overrides the `--databases` or `-B` option such that all name arguments following the option are regarded as table names.
`--use-frm`	For repair operations on MyISAM tables, get table structure from .frm file, so the table can be repaired even if the .MYI header is corrupted.
`-u, --user=name`	User for login if not current user.
`-v, --verbose`	Print info about the various stages. You can give this option several times to get even more information. See mysqlcheck and verbose, below.
`-V, --version`	Output version information and exit.
`--write-binlog`	Write ANALYZE, OPTIMIZE and REPAIR TABLE commands to the binary log. Enabled by default; use `--skip-write-binlog` when commands should not be sent to replication slaves.

Option Files

In addition to reading options from the command-line, `mysqlcheck` can also read options from option files. If an unknown option is provided to `mysqlcheck` in an option file, then it is ignored.

The following options relate to how MariaDB command-line tools handles option files. They must be given as the first argument on the command-line:

Option	Description
`--print-defaults`	Print the program argument list and exit.
`--no-defaults`	Don't read default options from any option file.
`--defaults-file=#`	Only read default options from the given file #.
`--defaults-extra-file=#`	Read this file after the global files are read.
`--defaults-group-suffix=#`	In addition to the default option groups, also read option groups with this suffix.

In MariaDB 10.2 and later, `mysqlcheck` is linked with MariaDB Connector/C. However, MariaDB Connector/C does not yet handle the parsing of option files for this client. That is still performed by the server option file parsing code. See MDEV-19035 for more information.

Option Groups

`mysqlcheck` reads options from the following option groups from option files:

Group	Description
`[mysqlcheck]`	Options read by `mysqlcheck`, which includes both MariaDB Server and MySQL Server.
`[mariadb-check]`	Options read by `mysqlcheck`. Available starting with MariaDB 10.4.6.
`[client]`	Options read by all MariaDB and MySQL client programs, which includes both MariaDB and MySQL clients. For example, `mysqldump`.
`[client-server]`	Options read by all MariaDB client programs and the MariaDB Server. This is useful for options like socket and port, which is common between the server and the clients.
`[client-mariadb]`	Options read by all MariaDB client programs.

Notes

Default Values

To see the default values for the options and also to see the arguments you get from configuration files you can do:

```
./client/mysqlcheck --print-defaults
./client/mysqlcheck --help
```

mysqlcheck and auto-repair

When running `mysqlcheck` with `--auto-repair` (as done by mysql_upgrade), `mysqlcheck` will first check all tables and then in a separate phase repair those that failed the check.

mysqlcheck and all-databases

`mysqlcheck --all-databases` will ignore the internal log tables general_log and slow_log as these can't be checked, repaired or optimized.

mysqlcheck and verbose

Using one `--verbose` option will give you more information about what mysqlcheck is doing.

Using two `--verbose` options will also give you connection information.

If you use three `--verbose` options you will also get, on stdout, all ALTER, RENAME, and CHECK commands that mysqlcheck executes.

1.1.2.1.12 OPTIMIZE TABLE

Syntax

```
OPTIMIZE [NO_WRITE_TO_BINLOG | LOCAL] TABLE
    tbl_name [, tbl_name] ...
    [WAIT n | NOWAIT]
```

Contents

Description

`OPTIMIZE TABLE` has two main functions. It can either be used to defragment tables, or to update the InnoDB fulltext index.

MariaDB starting with 10.3.0

WAIT/NOWAIT

Set the lock wait timeout. See WAIT and NOWAIT.

Defragmenting

`OPTIMIZE TABLE` works for InnoDB (before MariaDB 10.1.1, only if the innodb_file_per_table server system variable is set), Aria, MyISAM and ARCHIVE tables, and should be used if you have deleted a large part of a table or if you have made many changes to a 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.

This statement requires SELECT and INSERT privileges for the table.

By default, `OPTIMIZE TABLE` statements are written to the binary log and will be replicated. The `NO_WRITE_TO_BINLOG` keyword (`LOCAL` is an alias) will ensure the statement is not written to the binary log.

From MariaDB 10.3.19, `OPTIMIZE TABLE` statements are not logged to the binary log if read_only is set. See also Read-Only Replicas.

`OPTIMIZE TABLE` is also supported for partitioned tables. You can use `ALTER TABLE ... OPTIMIZE PARTITION` to optimize one or more partitions.

You can use `OPTIMIZE TABLE` to reclaim the unused space and to defragment the data file. With other storage engines, `OPTIMIZE TABLE` does nothing by default, and returns this message: " The storage engine for the table doesn't support optimize". However, if the server has been started with the `--skip-new` option, `OPTIMIZE TABLE` is linked to ALTER TABLE, and recreates the table. This operation frees the unused space and updates index statistics.

The Aria storage engine supports progress reporting for this statement.

If a MyISAM table is fragmented, concurrent inserts will not be performed until an `OPTIMIZE TABLE` statement is executed on that table, unless the concurrent_insert server system variable is set to `ALWAYS` .

Updating an InnoDB fulltext index

When rows are added or deleted to an InnoDB fulltext index, the index is not immediately re-organized, as this can be an expensive operation. Change statistics are stored in a separate location . The fulltext index is only fully re-organized when an `OPTIMIZE TABLE` statement is run.

By default, an OPTIMIZE TABLE will defragment a table. In order to use it to update fulltext index statistics, the innodb_optimize_fulltext_only system variable must be set to `1` . This is intended to be a temporary setting, and should be reset to `0` once the fulltext index has been re-organized.

Since fulltext re-organization can take a long time, the innodb_ft_num_word_optimize variable limits the re-organization to a number of words (2000 by default). You can run multiple OPTIMIZE statements to fully re-organize the index.

Defragmenting InnoDB tablespaces

MariaDB 10.1.1 merged the Facebook/Kakao defragmentation patch, allowing one to use `OPTIMIZE TABLE` to defragment InnoDB tablespaces. For this functionality to be enabled, the innodb_defragment system variable must be enabled. No new tables are created and there is no need to copy data from old tables to new tables. Instead, this feature loads `n` pages (determined by innodb-defragment-n-pages) and tries to move records so that pages would be full of records and then frees pages that are fully empty after the operation. Note that tablespace files (including ibdata1) will not shrink as the result of defragmentation, but one will get better memory utilization in the InnoDB buffer pool as there are fewer data pages in use.

See Defragmenting InnoDB Tablespaces for more details.

See Also

- Optimize Table in InnoDB with ALGORITHM set to INPLACE
- Optimize Table in InnoDB with ALGORITHM set to NOCOPY
- Optimize Table in InnoDB with ALGORITHM set to INSTANT

1.1.2.1.13 RENAME TABLE

Syntax

```
RENAME TABLE[S] [IF EXISTS] tbl_name
  [WAIT n | NOWAIT]
  TO new_tbl_name
    [, tbl_name2 TO new_tbl_name2] ...
```

Contents

Description

This statement renames one or more tables or views, but not the privileges associated with them.

IF EXISTS

MariaDB starting with 10.5.2

If this directive is used, one will not get an error if the table to be renamed doesn't exist.

The rename operation is done atomically, which means that no other session can access any of the tables while the rename is running. For example, if you have an existing table `old_table`, you can create another table `new_table` that has the same structure but is empty, and then replace the existing table with the empty one as follows (assuming that `backup_table` does not already exist):

```
CREATE TABLE new_table (...);
RENAME TABLE old_table TO backup_table, new_table TO old_table;
```

`tbl_name` can optionally be specified as `db_name.tbl_name`. See Identifier Qualifiers. This allows to use `RENAME` to move a table from a database to another (as long as they are on the same filesystem):

```
RENAME TABLE db1.t TO db2.t;
```

Note that moving a table to another database is not possible if it has some triggers. Trying to do so produces the following error:

```
ERROR 1435 (HY000): Trigger in wrong schema
```

Also, views cannot be moved to another database:

```
ERROR 1450 (HY000): Changing schema from 'old_db' to 'new_db' is not allowed.
```

Multiple tables can be renamed in a single statement. The presence or absence of the optional `S` (`RENAME TABLE` or `RENAME TABLES`) has no impact, whether a single or multiple tables are being renamed.

If a `RENAME TABLE` renames more than one table and one renaming fails, all renames executed by the same statement are rolled back.

Renames are always executed in the specified order. Knowing this, it is also possible to swap two tables' names:

```
RENAME TABLE t1 TO tmp_table,
    t2 TO t1,
    tmp_table TO t2;
```

WAIT/NOWAIT

MariaDB starting with 10.3.0

Set the lock wait timeout. See WAIT and NOWAIT.

Privileges

Executing the `RENAME TABLE` statement requires the DROP, CREATE and INSERT privileges for the table or the database.

Atomic RENAME TABLE

MariaDB starting with 10.6.1

From MariaDB 10.6, `RENAME TABLE` is atomic for most engines, including InnoDB, MyRocks, MyISAM and Aria (MDEV-23842). This means that if there is a crash (server down or power outage) during `RENAME TABLE` , all tables will revert to their original names and any changes to trigger files will be reverted.

In older MariaDB version there was a small chance that, during a server crash happening in the middle of `RENAME TABLE` , some tables could have been renamed (in the worst case partly) while others would not be renamed.

See Atomic DDL for more information.

1.1.2.1.14 REPAIR TABLE

Syntax

```
REPAIR [NO_WRITE_TO_BINLOG | LOCAL] TABLE
    tbl_name [, tbl_name] ...
    [QUICK] [EXTENDED] [USE_FRM]
```

Description

`REPAIR TABLE` repairs a possibly corrupted table. By default, it has the same effect as

```
myisamchk --recover tbl_name
```

or

```
aria_chk --recover tbl_name
```

See aria_chk and myisamchk for more.

`REPAIR TABLE` works for Archive, Aria, CSV and MyISAM tables. For InnoDB, see recovery modes. For CSV, see also Checking and Repairing CSV Tables. For Archive, this statement also improves compression. If the storage engine does not support this statement, a warning is issued.

This statement requires SELECT and INSERT privileges for the table.

By default, `REPAIR TABLE` statements are written to the binary log and will be replicated. The `NO_WRITE_TO_BINLOG` keyword (`LOCAL` is an alias) will ensure the statement is not written to the binary log.

From MariaDB 10.3.19, `REPAIR TABLE` statements are not logged to the binary log if read_only is set. See also Read-Only Replicas.

When an index is recreated, the storage engine may use a configurable buffer in the process. Incrementing the buffer speeds up the index

creation. Aria and MyISAM allocate a buffer whose size is defined by `aria_sort_buffer_size` or `myisam_sort_buffer_size`, also used for `ALTER TABLE`.

`REPAIR TABLE` is also supported for partitioned tables. However, the `USE_FRM` option cannot be used with this statement on a partitioned table.

`ALTER TABLE ... REPAIR PARTITION` can be used to repair one or more partitions.

The Aria storage engine supports progress reporting for this statement.

1.1.2.1.15 REPAIR VIEW

Syntax

```
REPAIR [NO_WRITE_TO_BINLOG | LOCAL] VIEW  view_name[, view_name] ... [FROM MYSQL]
```

Contents

Description

The `REPAIR VIEW` statement was introduced to assist with fixing MDEV-6916, an issue introduced in MariaDB 5.2 where the view algorithms were swapped compared to their MySQL on disk representation. It checks whether the view algorithm is correct. It is run as part of mysql_upgrade, and should not normally be required in regular use.

By default it corrects the checksum and if necessary adds the mariadb-version field. If the optional `FROM MYSQL` clause is used, and no mariadb-version field is present, the MERGE and TEMPTABLE algorithms are toggled.

By default, `REPAIR VIEW` statements are written to the binary log and will be replicated. The `NO_WRITE_TO_BINLOG` keyword (`LOCAL` is an alias) will ensure the statement is not written to the binary log.

See Also

- CHECK VIEW

1.1.2.1.16 REPLACE

Syntax

```
REPLACE [LOW_PRIORITY | DELAYED]
 [INTO] tbl_name [PARTITION (partition_list)] [(col,...)]
 {VALUES | VALUE} ({expr | DEFAULT},...),(...),...
[RETURNING select_expr
      [, select_expr ...]]
```

Or:

```
REPLACE [LOW_PRIORITY | DELAYED]
    [INTO] tbl_name [PARTITION (partition_list)]
    SET col={expr | DEFAULT}, ...
[RETURNING select_expr
      [, select_expr ...]]
```

Or:

```
REPLACE [LOW_PRIORITY | DELAYED]
    [INTO] tbl_name [PARTITION (partition_list)] [(col,...)]
    SELECT ...
[RETURNING select_expr
      [, select_expr ...]]
```

Contents

Description

`REPLACE` works exactly like `INSERT`, except that if an old row in the table has the same value as a new row for a `PRIMARY KEY` or a `UNIQUE` index, the old row is deleted before the new row is inserted. If the table has more than one `UNIQUE` keys, it is possible that the new row conflicts with more than one row. In this case, all conflicting rows will be deleted.

The table name can be specified in the form `db_name.tbl_name` or, if a default database is selected, in the form `tbl_name` (see Identifier Qualifiers). This allows to use `REPLACE ... SELECT` to copy rows between different databases.

MariaDB starting with 10.5.0

The RETURNING clause was introduced in MariaDB 10.5.0

Basically it works like this:

```
BEGIN;
SELECT 1 FROM t1 WHERE key=# FOR UPDATE;
IF found-row
  DELETE FROM t1 WHERE key=# ;
ENDIF
INSERT INTO t1 VALUES (...);
END;
```

The above can be replaced with:

```
REPLACE INTO t1 VALUES (...)
```

`REPLACE` is a MariaDB/MySQL extension to the SQL standard. It either inserts, or deletes and inserts. For other MariaDB/MySQL extensions to standard SQL --- that also handle duplicate values --- see IGNORE and INSERT ON DUPLICATE KEY UPDATE.

Note that unless the table has a `PRIMARY KEY` or `UNIQUE` index, using a `REPLACE` statement makes no sense. It becomes equivalent to `INSERT`, because there is no index to be used to determine whether a new row duplicates another.

Values for all columns are taken from the values sSee Partition Pruning and Selection for details.pecified in the `REPLACE` statement. Any missing columns are set to their default values, just as happens for `INSERT`. You cannot refer to values from the current row and use them in the new row. If you use an assignment such as `'SET col = col + 1'`, the reference to the column name on the right hand side is treated as `DEFAULT(col)`, so the assignment is equivalent to `'SET col = DEFAULT(col) + 1'`.

To use `REPLACE`, you must have both the `INSERT` and `DELETE` privileges for the table.

There are some gotchas you should be aware of, before using `REPLACE`:

- If there is an `AUTO_INCREMENT` field, a new value will be generated.
- If there are foreign keys, `ON DELETE` action will be activated by `REPLACE`.
- Triggers on `DELETE` and `INSERT` will be activated by `REPLACE`.

To avoid some of these behaviors, you can use `INSERT ... ON DUPLICATE KEY UPDATE`.

This statement activates INSERT and DELETE triggers. See Trigger Overview for details.

PARTITION

See Partition Pruning and Selection for details.

REPLACE RETURNING

`REPLACE ... RETURNING` returns a resultset of the replaced rows.

This returns the listed columns for all the rows that are replaced, or alternatively, the specified SELECT expression. Any SQL expressions which can be calculated can be used in the select expression for the RETURNING clause, including virtual columns and aliases, expressions which use various operators such as bitwise, logical and arithmetic operators, string functions, date-time functions, numeric functions, control flow functions, secondary functions and stored functions. Along with this, statements which have subqueries and prepared statements can also be used.

Examples

Simple REPLACE statement

```
REPLACE INTO t2 VALUES (1,'Leopard'),(2,'Dog') RETURNING id2, id2+id2
as Total ,id2|id2, id2&&id2;
+-----+-------+---------+----------+
| id2 | Total | id2|id2 | id2&&id2 |
+-----+-------+---------+----------+
|   1 |     2 |       1 |        1 |
|   2 |     4 |       2 |        1 |
+-----+-------+---------+----------+
```

Using stored functions in RETURNING

```
DELIMITER |
CREATE FUNCTION f(arg INT) RETURNS INT
    BEGIN
      RETURN (SELECT arg+arg);
    END|

DELIMITER ;
PREPARE stmt FROM "REPLACE INTO t2 SET id2=3, animal2='Fox' RETURNING f2(id2),
UPPER(animal2)";

EXECUTE stmt;
+---------+----------------+
| f2(id2) | UPPER(animal2) |
+---------+----------------+
|       6 | FOX            |
+---------+----------------+
```

Subqueries in the statement

```
REPLACE INTO t1 SELECT * FROM t2 RETURNING (SELECT id2 FROM t2 WHERE
id2 IN (SELECT id2 FROM t2 WHERE id2=1)) AS new_id;
+--------+
| new_id |
+--------+
|      1 |
|      1 |
|      1 |
|      1 |
+--------+
```

Subqueries in the RETURNING clause that return more than one row or column cannot be used..

Aggregate functions cannot be used in the RETURNING clause. Since aggregate functions work on a set of values and if the purpose is to get the row count, ROW_COUNT() with SELECT can be used, or it can be used in REPLACE...SEL== Description

`REPLACE ... RETURNING` returns a resultset of the replaced rows.

This returns the listed columns for all the rows that are replaced, or alternatively, the specified SELECT expression. Any SQL expressions which can be calculated can be used in the select expression for the RETURNING clause, including virtual columns and aliases, expressions which use various operators such as bitwise, logical and arithmetic operators, string functions, date-time functions, numeric functions, control flow functions, secondary functions and stored functions. Along with this, statements which have subqueries and prepared statements can also be used.

Examples

Simple REPLACE statement

```
REPLACE INTO t2 VALUES (1,'Leopard'),(2,'Dog') RETURNING id2, id2+id2
as Total ,id2|id2, id2&&id2;
+-----+-------+---------+----------+
| id2 | Total | id2|id2 | id2&&id2 |
+-----+-------+---------+----------+
|   1 |     2 |       1 |        1 |
|   2 |     4 |       2 |        1 |
+-----+-------+---------+----------+
```

Using stored functions in RETURNING

```
DELIMITER |
CREATE FUNCTION f(arg INT) RETURNS INT
    BEGIN
      RETURN (SELECT arg+arg);
    END|

DELIMITER ;
PREPARE stmt FROM "REPLACE INTO t2 SET id2=3, animal2='Fox' RETURNING f2(id2),
UPPER(animal2)";

EXECUTE stmt;
+---------+----------------+
| f2(id2) | UPPER(animal2) |
+---------+----------------+
|       6 | FOX            |
+---------+----------------+
```

Subqueries in the statement

```
REPLACE INTO t1 SELECT * FROM t2 RETURNING (SELECT id2 FROM t2 WHERE
id2 IN (SELECT id2 FROM t2 WHERE id2=1)) AS new_id;
+--------+
| new_id |
+--------+
|      1 |
|      1 |
|      1 |
|      1 |
+--------+
```

Subqueries in the RETURNING clause that return more than one row or column cannot be used..

Aggregate functions cannot be used in the RETURNING clause. Since aggregate functions work on a set of values and if the purpose is to get the row count, ROW_COUNT() with SELECT can be used, or it can be used in REPLACE...SELECT...RETURNING if the table in the RETURNING clause is not the same as the REPLACE table. ECT...RETURNING if the table in the RETURNING clause is not the same as the REPLACE table.

See Also

- INSERT
- HIGH_PRIORITY and LOW_PRIORITY clauses
- INSERT DELAYED for details on the `DELAYED` clause

1.1.2.1.17 SHOW COLUMNS

Syntax

```
SHOW [FULL] {COLUMNS | FIELDS} FROM tbl_name [FROM db_name]
    [LIKE 'pattern' | WHERE expr]
```

Contents

Description

`SHOW COLUMNS` displays information about the columns in a given table. It also works for views. The `LIKE` clause, if present on its own, indicates which column names to match. The `WHERE` and `LIKE` clauses can be given to select rows using more general conditions, as discussed in Extended SHOW.

If the data types differ from what you expect them to be based on a `CREATE TABLE` statement, note that MariaDB sometimes changes data types when you create or alter a table. The conditions under which this occurs are described in the Silent Column Changes article.

The `FULL` keyword causes the output to include the column collation and comments, as well as the privileges you have for each column.

You can use `db_name.tbl_name` as an alternative to the `tbl_name FROM db_name` syntax. In other words, these two statements are equivalent:

```
SHOW COLUMNS FROM mytable FROM mydb;
SHOW COLUMNS FROM mydb.mytable;
```

`SHOW COLUMNS` displays the following values for each table column:

Field indicates the column name.

Type indicates the column data type.

Collation indicates the collation for non-binary string columns, or NULL for other columns. This value is displayed only if you use the FULL keyword.

The **Null** field contains YES if NULL values can be stored in the column, NO if not.

The **Key** field indicates 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, non-unique 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-valued index that cannot contain `NULL` values.
- If **Key** is ***MUL***, multiple occurrences of a given value are allowed within the column. The column is the first column of a non-unique index or a unique-valued index that can contain `NULL` values.

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.

The **Default** field indicates the default value that is assigned to the column.

The **Extra** field contains any additional information that is available about a given column.

Value	Description
`AUTO_INCREMENT`	The column was created with the `AUTO_INCREMENT` keyword.
`PERSISTENT`	The column was created with the `PERSISTENT` keyword. (New in 5.3)
`VIRTUAL`	The column was created with the `VIRTUAL` keyword. (New in 5.3)
on update `CURRENT_TIMESTAMP`	The column is a `TIMESTAMP` column that is automatically updated on `INSERT` and `UPDATE`.

Privileges indicates the privileges you have for the column. This value is displayed only if you use the `FULL` keyword.

Comment indicates any comment the column has. This value is displayed only if you use the `FULL` keyword.

`SHOW FIELDS` is a synonym for `SHOW COLUMNS`. Also `DESCRIBE` and `EXPLAIN` can be used as shortcuts.

You can also list a table's columns with:

```
mysqlshow db_name tbl_name
```

See the `mysqlshow` command for more details.

The `DESCRIBE` statement provides information similar to `SHOW COLUMNS`. The `information_schema.COLUMNS` table provides similar, but more complete, information.

The `SHOW CREATE TABLE`, `SHOW TABLE STATUS`, and `SHOW INDEX` statements also provide information about tables.

Examples

```
SHOW COLUMNS FROM city;
+------------+----------+------+-----+---------+----------------+
| Field      | Type     | Null | Key | Default | Extra          |
+------------+----------+------+-----+---------+----------------+
| Id         | int(11)  | NO   | PRI | NULL    | auto_increment |
| Name       | char(35) | NO   |     |         |                |
| Country    | char(3)  | NO   | UNI |         |                |
| District   | char(20) | YES  | MUL |         |                |
| Population | int(11)  | NO   |     | 0       |                |
+------------+----------+------+-----+---------+----------------+
```

```
SHOW COLUMNS FROM employees WHERE Type LIKE 'Varchar%';
+---------------+-------------+------+-----+---------+-------+
| Field         | Type        | Null | Key | Default | Extra |
+---------------+-------------+------+-----+---------+-------+
| first_name    | varchar(30) | NO   | MUL | NULL    |       |
| last_name     | varchar(40) | NO   |     | NULL    |       |
| position      | varchar(25) | NO   |     | NULL    |       |
| home_address  | varchar(50) | NO   |     | NULL    |       |
| home_phone    | varchar(12) | NO   |     | NULL    |       |
| employee_code | varchar(25) | NO   | UNI | NULL    |       |
+---------------+-------------+------+-----+---------+-------+
```

See Also

- DESCRIBE
- mysqlshow
- SHOW CREATE TABLE
- SHOW TABLE STATUS
- SHOW INDEX
- Extended SHOW
- Silent Column Changes

1.1.2.1.18 SHOW CREATE TABLE

Syntax

```
SHOW CREATE TABLE tbl_name
```

Contents

Description

Shows the CREATE TABLE statement that created the given table. The statement requires the SELECT privilege for the table. This statement also works with views and SEQUENCE.

`SHOW CREATE TABLE` quotes table and column names according to the value of the sql_quote_show_create server system variable.

Certain SQL_MODE values can result in parts of the original CREATE statement not being included in the output. MariaDB-specific table options, column options, and index options are not included in the output of this statement if the NO_TABLE_OPTIONS, NO_FIELD_OPTIONS and NO_KEY_OPTIONS SQL_MODE flags are used. All MariaDB-specific table attributes are also not shown when a non-MariaDB/MySQL emulation mode is used, which includes ANSI, DB2, POSTGRESQL, MSSQL, MAXDB or ORACLE.

Invalid table options, column options and index options are normally commented out (note, that it is possible to create a table with invalid options, by altering a table of a different engine, where these options were valid). To have them uncommented, enable the IGNORE_BAD_TABLE_OPTIONS SQL_MODE. Remember that replaying a CREATE TABLE statement with uncommented invalid options will fail with an error, unless the IGNORE_BAD_TABLE_OPTIONS SQL_MODE is in effect.

Note that `SHOW CREATE TABLE` is not meant to provide metadata about a table. It provides information about how the table was declared, but the real table structure could differ a bit. For example, if an index has been declared as `HASH`, the `CREATE TABLE` statement returned by `SHOW CREATE TABLE` will declare that index as `HASH`; however, it is possible that the index is in fact a `BTREE`, because the storage engine does not support `HASH`.

MariaDB starting with 10.2.1

MariaDB 10.2.1 permits TEXT and BLOB data types to be assigned a DEFAULT value. As a result, from MariaDB 10.2.1, `SHOW CREATE TABLE` will append a `DEFAULT NULL` to nullable TEXT or BLOB fields if no specific default is provided.

MariaDB starting with 10.2.2

From MariaDB 10.2.2, numbers are no longer quoted in the `DEFAULT` clause in `SHOW CREATE` statement. Previously, MariaDB quoted numbers.

Examples

```
SHOW CREATE TABLE t\G
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `s` char(60) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

With sql_quote_show_create off:

```
SHOW CREATE TABLE t\G
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE t (
  id int(11) NOT NULL AUTO_INCREMENT,
  s char(60) DEFAULT NULL,
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Unquoted numeric DEFAULTs, from MariaDB 10.2.2:

```
CREATE TABLE td (link TINYINT DEFAULT 1);

SHOW CREATE TABLE td\G
*************************** 1. row ***************************
       Table: td
Create Table: CREATE TABLE `td` (
  `link` tinyint(4) DEFAULT 1
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Quoted numeric DEFAULTs, until MariaDB 10.2.1:

```
CREATE TABLE td (link TINYINT DEFAULT 1);

SHOW CREATE TABLE td\G
*************************** 1. row ***************************
       Table: td
Create Table: CREATE TABLE `td` (
  `link` tinyint(4) DEFAULT '1'
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

SQL_MODE impacting the output:

```
SELECT @@sql_mode;
+-------------------------------------------------------------------------------------------+
| @@sql_mode                                                                                |
+-------------------------------------------------------------------------------------------+
| STRICT_TRANS_TABLES,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+-------------------------------------------------------------------------------------------+

CREATE TABLE `t1` (
       `id` int(11) NOT NULL AUTO_INCREMENT,
       `msg` varchar(100) DEFAULT NULL,
       PRIMARY KEY (`id`)
     ) ENGINE=InnoDB DEFAULT CHARSET=latin1
;

SHOW CREATE TABLE t1\G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `msg` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1

SET SQL_MODE=ORACLE;

SHOW CREATE TABLE t1\G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE "t1" (
  "id" int(11) NOT NULL,
  "msg" varchar(100) DEFAULT NULL,
  PRIMARY KEY ("id")
```

See Also

- SHOW CREATE SEQUENCE
- SHOW CREATE VIEW

1.1.2.1.19 SHOW INDEX

Syntax

```
SHOW {INDEX | INDEXES | KEYS}
 FROM tbl_name [FROM db_name]
 [WHERE expr]
```

Contents

Description

`SHOW INDEX` returns table index information. The format resembles that of the SQLStatistics call in ODBC.

You can use `db_name.tbl_name` as an alternative to the `tbl_name FROM db_name` syntax. These two statements are equivalent:

```
SHOW INDEX FROM mytable FROM mydb;
SHOW INDEX FROM mydb.mytable;
```

`SHOW KEYS` and `SHOW INDEXES` are synonyms for `SHOW INDEX`.

You can also list a table's indexes with the mariadb-show/mysqlshow command:

```
mysqlshow -k db_name tbl_name
```

The information_schema.STATISTICS table stores similar information.

The following fields are returned by `SHOW INDEX`.

Field	Description
`Table`	Table name
`Non_unique`	`1` if the index permits duplicate values, `0` if values must be unique.
`Key_name`	Index name. The primary key is always named `PRIMARY`.
`Seq_in_index`	The column's sequence in the index, beginning with `1`.
`Column_name`	Column name.
`Collation`	Either `A`, if the column is sorted in ascending order in the index, or `NULL` if it's not sorted.
`Cardinality`	Estimated number of unique values in the index. The cardinality statistics are calculated at various times, and can help the optimizer make improved decisions.
`Sub_part`	`NULL` if the entire column is included in the index, or the number of included characters if not.
`Packed`	`NULL` if the index is not packed, otherwise how the index is packed.
`Null`	`NULL` if `NULL` values are permitted in the column, an empty string if `NULL`'s are not permitted.
`Index_type`	The index type, which can be `BTREE`, `FULLTEXT`, `HASH` or `RTREE`. See Storage Engine Index Types.
`Comment`	Other information, such as whether the index is disabled.
`Index_comment`	Contents of the `COMMENT` attribute when the index was created.
`Ignored`	Whether or not an index will be ignored by the optimizer. See Ignored Indexes. From MariaDB 10.6.0.

The `WHERE` and `LIKE` clauses can be given to select rows using more general conditions, as discussed in Extended SHOW.

Examples

```
CREATE TABLE IF NOT EXISTS `employees_example` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `first_name` varchar(30) NOT NULL,
  `last_name` varchar(40) NOT NULL,
  `position` varchar(25) NOT NULL,
  `home_address` varchar(50) NOT NULL,
  `home_phone` varchar(12) NOT NULL,
  `employee_code` varchar(25) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `employee_code` (`employee_code`),
  KEY `first_name` (`first_name`,`last_name`)
) ENGINE=Aria;

INSERT INTO `employees_example` (`first_name`, `last_name`, `position`, `home_address`, `home_phone`, `employee_code`)
  VALUES
  ('Mustapha', 'Mond', 'Chief Executive Officer', '692 Promiscuous Plaza', '326-555-3492', 'MM1'),
  ('Henry', 'Foster', 'Store Manager', '314 Savage Circle', '326-555-3847', 'HF1'),
  ('Bernard', 'Marx', 'Cashier', '1240 Ambient Avenue', '326-555-8456', 'BM1'),
  ('Lenina', 'Crowne', 'Cashier', '281 Bumblepuppy Boulevard', '328-555-2349', 'LC1'),
  ('Fanny', 'Crowne', 'Restocker', '1023 Bokanovsky Lane', '326-555-6329', 'FC1'),
  ('Helmholtz', 'Watson', 'Janitor', '944 Soma Court', '329-555-2478', 'HW1');
```

```
SHOW INDEXES FROM employees_example\G
*************************** 1. row ***************************
        Table: employees_example
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: id
    Collation: A
  Cardinality: 6
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
      Ignored: NO
*************************** 2. row ***************************
        Table: employees_example
   Non_unique: 0
     Key_name: employee_code
 Seq_in_index: 1
  Column_name: employee_code
    Collation: A
  Cardinality: 6
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
      Ignored: NO
*************************** 3. row ***************************
        Table: employees_example
   Non_unique: 1
     Key_name: first_name
 Seq_in_index: 1
  Column_name: first_name
    Collation: A
  Cardinality: NULL
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
      Ignored: NO
*************************** 4. row ***************************
        Table: employees_example
   Non_unique: 1
     Key_name: first_name
 Seq_in_index: 2
  Column_name: last_name
    Collation: A
  Cardinality: NULL
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
      Ignored: NO
```

See Also

- Ignored Indexes

1.1.2.1.20 TRUNCATE TABLE

Syntax

```
TRUNCATE [TABLE] tbl_name
  [WAIT n | NOWAIT]
```

Contents

Description

`TRUNCATE TABLE` empties a table completely. It requires the `DROP` privilege. See GRANT.

`tbl_name` can also be specified in the form `db_name.tbl_name` (see Identifier Qualifiers).

Logically, `TRUNCATE TABLE` is equivalent to a DELETE statement that deletes all rows, but there are practical differences under some circumstances.

`TRUNCATE TABLE` will fail for an InnoDB table if any FOREIGN KEY constraints from other tables reference the table, returning the error:

```
ERROR 1701 (42000): Cannot truncate a table referenced in a foreign key constraint
```

Foreign Key constraints between columns in the same table are permitted.

For an InnoDB table, if there are no `FOREIGN KEY` constraints, InnoDB performs fast truncation by dropping the original table and creating an empty one with the same definition, which is much faster than deleting rows one by one. The AUTO_INCREMENT counter is reset by `TRUNCATE TABLE`, regardless of whether there is a `FOREIGN KEY` constraint.

The count of rows affected by `TRUNCATE TABLE` is accurate only when it is mapped to a `DELETE` statement.

For other storage engines, `TRUNCATE TABLE` differs from `DELETE` in the following ways:

- Truncate operations drop and re-create the table, which is much faster than deleting rows one by one, particularly for large tables.
- Truncate operations cause an implicit commit.
- Truncation operations cannot be performed if the session holds an active table lock.
- Truncation operations do not return a meaningful value for the number of deleted rows. The usual result is "0 rows affected," which should be interpreted as "no information."
- As long as the table format file `tbl_name.frm` is valid, the table can be re-created as an empty table with `TRUNCATE TABLE`, even if the data or index files have become corrupted.
- The table handler does not remember the last used AUTO_INCREMENT value, but starts counting from the beginning. This is true even for MyISAM and InnoDB, which normally do not reuse sequence values.
- When used with partitioned tables, `TRUNCATE TABLE` preserves the partitioning; that is, the data and index files are dropped and re-created, while the partition definitions (.par) file is unaffected.
- Since truncation of a table does not make any use of `DELETE`, the `TRUNCATE` statement does not invoke `ON DELETE` triggers.
- `TRUNCATE TABLE` will only reset the values in the Performance Schema summary tables to zero or null, and will not remove the rows.

For the purposes of binary logging and replication, `TRUNCATE TABLE` is treated as DROP TABLE followed by CREATE TABLE (DDL rather than DML).

`TRUNCATE TABLE` does not work on views. Currently, `TRUNCATE TABLE` drops all historical records from a system-versioned table.

MariaDB starting with 10.3.0

WAIT/NOWAIT

Set the lock wait timeout. See WAIT and NOWAIT.

Oracle-mode

Oracle-mode from MariaDB 10.3 permits the optional keywords REUSE STORAGE or DROP STORAGE to be used.

```
TRUNCATE [TABLE] tbl_name [{DROP | REUSE} STORAGE] [WAIT n | NOWAIT]
```

These have no effect on the operation.

Performance

`TRUNCATE TABLE` is faster than DELETE, because it drops and re-creates a table.

With InnoDB, `TRUNCATE TABLE` is slower if innodb_file_per_table=ON is set (the default). This is because `TRUNCATE TABLE` unlinks the underlying tablespace file, which can be an expensive operation. See MDEV-8069 for more details.

The performance issues with innodb_file_per_table=ON can be exacerbated in cases where the InnoDB buffer pool is very large and innodb_adaptive_hash_index=ON is set. In that case, using DROP TABLE followed by CREATE TABLE instead of `TRUNCATE TABLE` may perform better. Setting innodb_adaptive_hash_index=OFF (it defaults to ON before MariaDB 10.5) can also help. In MariaDB 10.2 only, from MariaDB 10.2.19, this performance can also be improved by setting innodb_safe_truncate=OFF. See MDEV-9459 for more details.

Setting innodb_adaptive_hash_index=OFF can also improve `TRUNCATE TABLE` performance in general. See MDEV-16796 for more details.

See Also

- TRUNCATE function
- innodb_safe_truncate system variable
- Oracle mode from MariaDB 10.3

1.1.2.1.21 UPDATE

Syntax

Single-table syntax:

```
UPDATE [LOW_PRIORITY] [IGNORE] table_reference
  [PARTITION (partition_list)]
  [FOR PORTION OF period FROM expr1 TO expr2]
  SET col1={expr1|DEFAULT} [,col2={expr2|DEFAULT}] ...
  [WHERE where_condition]
  [ORDER BY ...]
  [LIMIT row_count]
```

Multiple-table syntax:

```
UPDATE [LOW_PRIORITY] [IGNORE] table_references
    SET col1={expr1|DEFAULT} [, col2={expr2|DEFAULT}] ...
    [WHERE where_condition]
```

Contents

Description

For the single-table syntax, the `UPDATE` statement updates columns of existing rows in the named table with new values. The `SET` clause indicates which columns to modify and the values they should be given. Each value can be given as an expression, or the keyword `DEFAULT` to set a column explicitly to its default value. The `WHERE` clause, if given, specifies the conditions that identify which rows to update. With no `WHERE` clause, all rows are updated. If the ORDER BY clause is specified, the rows are updated in the order that is specified. The LIMIT clause places a limit on the number of rows that can be updated.

Until MariaDB 10.3.2, for the multiple-table syntax, `UPDATE` updates rows in each table named in table_references that satisfy the conditions. In this case, ORDER BY and LIMIT cannot be used. This restriction was lifted in MariaDB 10.3.2 and both clauses can be used with multiple-table updates. An `UPDATE` can also reference tables which are located in different databases; see Identifier Qualifiers for the syntax.

`where_condition` is an expression that evaluates to true for each row to be updated.

`table_references` and `where_condition` are as specified as described in SELECT.

For single-table updates, assignments are evaluated in left-to-right order, while for multi-table updates, there is no guarantee of a particular order. If the `SIMULTANEOUS_ASSIGNMENT` sql_mode (available from MariaDB 10.3.5) is set, UPDATE statements evaluate all assignments simultaneously.

You need the `UPDATE` privilege only for columns referenced in an `UPDATE` that are actually updated. You need only the SELECT privilege for any columns that are read but not modified. See GRANT.

The `UPDATE` statement supports the following modifiers:

- If you use the `LOW_PRIORITY` keyword, execution of the `UPDATE` is delayed until no other clients are reading from the table. This affects only storage engines that use only table-level locking (MyISAM, MEMORY, MERGE). See HIGH_PRIORITY and LOW_PRIORITY clauses for details.
- If you use the `IGNORE` keyword, the update statement does not abort even if errors occur during the update. Rows for which duplicate-key conflicts occur are not updated. Rows for which columns are updated to values that would cause data conversion errors are updated to the closest valid values instead.

PARTITION

See Partition Pruning and Selection for details.

FOR PORTION OF

MariaDB starting with 10.4.3

See Application Time Periods - Updating by Portion.

UPDATE Statements With the Same Source and Target

MariaDB starting with 10.3.2

From MariaDB 10.3.2, UPDATE statements may have the same source and target.

For example, given the following table:

```
DROP TABLE t1;
CREATE TABLE t1 (c1 INT, c2 INT);
INSERT INTO t1 VALUES (10,10), (20,20);
```

Until MariaDB 10.3.1, the following UPDATE statement would not work:

```
UPDATE t1 SET c1=c1+1 WHERE c2=(SELECT MAX(c2) FROM t1);
ERROR 1093 (HY000): Table 't1' is specified twice,
  both as a target for 'UPDATE' and as a separate source for data
```

From MariaDB 10.3.2, the statement executes successfully:

```
UPDATE t1 SET c1=c1+1 WHERE c2=(SELECT MAX(c2) FROM t1);

SELECT * FROM t1;
+------+------+
| c1   | c2   |
+------+------+
|   10 |   10 |
|   21 |   20 |
+------+------+
```

Example

Single-table syntax:

```
UPDATE table_name SET column1 = value1, column2 = value2 WHERE id=100;
```

Multiple-table syntax:

```
UPDATE tab1, tab2 SET tab1.column1 = value1, tab1.column2 = value2 WHERE tab1.id = tab2.id;
```

See Also

- How IGNORE works
- SELECT
- ORDER BY
- LIMIT
- Identifier Qualifiers