From the MariaDB Knowledge Base

2022-03

MariaDB Server Documentation

MariaDB Foundation

Ian Gilfillan (Editor)

MariaDB Server Documentation

MariaDB Knowledge Base

For any errors please see Bug Reporting

Document generated on: 2022-03-30

Chapter Contents

Table of Contents

H3Dr1 Using MariaDB Server

Documentation on using MariaDB Server.

SQL Statements & Structure
SQL statements, structure, and rules.

Built-in Functions
Functions and procedures in MariaDB.

Clients & Utilities
Client and utility programs for MariaDB.

H3Dr1.1 SQL Statements & Structure

The letters *SQL* stand for Structured Query Language. As with all languages—even computer languages—there are grammar rules. This includes a certain structure to statements, acceptable punctuation (i.e., operators and delimiters), and a vocabulary (i.e., reserve words).

SQL Statements
Explanations of all of the MariaDB SQL statements.

SQL Language Structure
Explanation of SQL grammar rules, including reserved words and literals.

Geographic & Geometric Features
Spatial extensions for geographic and geometric features.

NoSQL
NoSQL-related commands and interfaces

Operators
Operators for comparing and assigning values.

Sequences
Sequence objects, an alternative to AUTO_INCREMENT.

Temporal Tables
MariaDB supports system-versioning, application-time periods and bitemporal tables.

There are 9 related questions.

H3Dr1.1.1 SQL Statements

Complete list of SQL statements for data definition, data manipulation, etc.

Account Management SQL Commands
CREATE/DROP USER, GRANT, REVOKE, SET PASSWORD etc.

Administrative SQL Statements
SQL statements for setting, flushing and displaying server variables and resources.

Data Definition
SQL commands for defining data, such as ALTER, CREATE, DROP, RENAME etc.

Data Manipulation
SQL commands for querying and manipulating data, such as SELECT, UPDATE, DELETE etc.

Prepared Statements
Prepared statements from any client using the text based prepared statement interface.

Programmatic & Compound Statements
Compound SQL statements for stored routines and in general.

Stored Routine Statements
SQL statements related to creating and using stored routines.

Table Statements
Documentation on creating, altering, analyzing and maintaining tables.

Transactions
Sequence of statements that are either completely successful, or have no effect on any schemas

HELP Command
The HELP command will retrieve syntax and help within the mysql client.

Comment Syntax
Comment syntax and style.

Built-in Functions
Functions and procedures in MariaDB.

There are 16 related questions.

H3Dr1.1.1.1 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.

H3Dr1.1.1.1.1 CREATE

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.

H3Dr1.1.1.1.2 ALTER

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.

H3Dr1.1.1.1.3 DROP

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

H3Dr1.1.1.1.4 G

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 OPTION |
| 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 OPTION |
| 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.

H3Dr1.1.1.1.5 RENAME

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';
```

H3Dr1.1.1.1.6 RE

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
```

H3Dr1.1.1.1.7 SET PASS

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

H3Dr1.1.1.1.8 CREATE

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

H3Dr1.1.1.1.9 DROP

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

H3Dr1.1.1.1.10 SET

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           |
+----------------+
```

H3Dr1.1.1.1.11 SET DEFAULT

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;
```

H3Dr1.1.1.1.12 SHOW GR

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

H3Dr1.1.1.1.13 SHOW CREATE

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.

H3Dr1.1.1.2 Administrative SQL Statements

SQL statements for administering MariaDB.

Table Statements
Documentation on creating, altering, analyzing and maintaining tables.

ANALYZE and EXPLAIN Statements
Articles on the ANALYZE and EXPLAIN statements

BACKUP Commands
Commands used by backup tools.

FLUSH Commands
Commands to flush or reset various caches in MariaDB.

Replication Commands
List of replication-related commands.

Plugin SQL Statements
List of SQL statements related to plugins.

SET Commands
The SET commands

SHOW
Articles on the various SHOW commands.

System Tables

BINLOG
Generated by mysqlbinlog

PURGE BINARY LOGS
PURGE BINARY LOGS removes all binary logs from the server, prior to the provided date or log file.

CACHE INDEX
Caches MyISAM or Aria indexes

DESCRIBE
Information about columns in a table.

EXECUTE Statement
Executes a previously PREPAREd statement

HELP Command
The HELP command will retrieve syntax and help within the mysql client.

KILL [CONNECTION | QUERY]
Kill connection by query or thread id.

LOAD INDEX
Loads one or more indexes from one or more MyISAM/Aria tables into a key buffer

RESET
Overall description of the different RESET commands

SHUTDOWN
Shuts down the server.

USE
Set the current default database.

There are 1 related questions.

H3Dr1.1.1.2.1 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.

H3Dr1.1.1.2.1.1 ALTER

This category is for documentation on the various ALTER statements.

ALTER TABLE
Modify a table's definition.

ALTER DATABASE

Change the overall characteristics of a database.

ALTER EVENT

Change an existing event.

ALTER FUNCTION

Change the characteristics of a stored function.

ALTER LOGFILE GROUP

Only useful with MySQL Cluster, and has no effect in MariaDB.

ALTER PROCEDURE

Change stored procedure characteristics.

ALTER SEQUENCE

Change options for a SEQUENCE.

ALTER SERVER

Updates mysql.servers table.

ALTER TABLESPACE

ALTER TABLESPACE is not available in MariaDB.

ALTER USER

Modify an existing MariaDB account.

ALTER VIEW

Change a view definition.

There are 1 related questions.

H3Dr1.1.1.2.1.1.1 ALTER T

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.

This statement is equivalent to the following:

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

See the ALGORITHM alter specification for more information.

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).

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

H3Dr1.1.1.2.1.1.2 ALTER DATA

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

H3Dr1.1.1.2.1.1.3 ALTER E

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

H3Dr1.1.1.2.1.1.4 ALTER FUNC

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

H3Dr1.1.1.2.1.1.5 ALTER LOGFILE G

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.