Some notes on MariaDB

John R Hudson[*]

November 2013

Contents

[*]The author would welcome notification of any errors or possible misunderstandings.

1 Preface

These notes only cover installing and using a single MariaDB server from the `.rpm` or `.deb` files of a Linux distribution. If you want to set up slave servers, compile from source or use a different operating system, look at the MariaDB documentation.

2 What is MariaDB

MariaDB is a fork of the MySQL database software which offers a ‘drop-in’ replacement for MySQL up to version 5.5 but will cease to do so with version 5.6 when MariaDB will be designated version 10 to distinguish it from MySQL.

MySQL AB was bought by Sun Microsystems for $1billion in 2008 and subsequently acquired by Oracle in 2010.

Development of MariaDB began in 2009 and has been overseen since 2013 by the MariaDB Foundation.

3 Starting the processes and setting the root password

First check that the MariaDB daemon is running; in openSuSE Linux go to YaST ▷ System Services and ensure that the `Enabled` column for the `mysqld` entry is set to `Yes`. If it is not, setting it to `Yes` will start `mysql` immediately and on every subsequent boot.

On Linux systems using `systemd`, you can check the status of the `mysqld` by entering:

```
systemctl status mysql.service
```

in a console as Linux `root` user.

If this is a new installation and you have no data preserved in a `mysqldump` backup file, run

```
mysql_secure_installation
```

This allows you to enter a root password and then, by answering `y` to each subsequent question, close all the loopholes that are left open to enable you to install MariaDB.

If you have a backup file created with `mysqldump`, ignore the above and follow the instructions in Backing up and restoring databases.

From here on, whenever you wish to log on as `root`, log on using

```
mysql -u root -p
```

and MariaDB will prompt you for your password which will not appear on the screen.

3.1 A note about root

`root` in MariaDB is not the same as `root` in Linux though it may be helpful for them to be the same person as some of the privileges which the MariaDB `root` has overlap those that the Linux `root` has. Consequently, the password which `root` has in MariaDB need not, and preferably should not be, the same password as the password which `root` has in Linux. Having different passwords means that, should someone succeed in gaining access to the site as Linux `root`, access to the MariaDB databases would still be protected by the MariaDB passwords. MariaDB passwords can be up to 16 characters long. (The MySQL help used to recommend that passwords should not be available in a dictionary and could be ‘encrypted’ by typing a sequence of characters using the keys to left or right of the ones used in a memorable sequence of characters or by using a mnemonic which yields a password which is not a recognisable word.)

3.2 A note about hosts

The default installation has two hosts: *localhost* and the HOST_NAME of your installation (in openSUSE this is a randomly generated hostname such as *linux-7k4h.site*). If you are logged on as MariaDB `root`, you can check this by entering

```
SELECT host, user FROM mysql.user;
```

In this statement `mysql` is the database and `user` the table which you are accessing. You will use *localhost* when accessing the databases locally and HOST_NAME when you access the databases remotely.

4 Connecting and issuing commands

Whenever you want to start MariaDB, open a terminal window and enter

```
mysql -u user_name -p
```

(until you have added some users, `root` will be your only user).

But, if you are completely new to MariaDB, it is worth using

```
mysql -U -u user_name -p
```

which protects you from inadvertently changing the values of all the rows in a column by forgetting to add a WHERE clause (see section 22 on page 24).

Either way MariaDB will prompt you for your password with

```
Enter password:
```

(Your password will not appear on the screen as you enter it.)

If you enter it successfully, you will see the prompt

```
mysql>
```

Thereafter, all commands except USE and QUIT must be terminated with ‘`;`’ except when requesting a vertical display using ‘`\G`’. You can enter multiple commands terminated with ‘`;`’ on the same line or multiple line commands terminated with ‘`;`’. If you enter a multiple line command the prompt may become `>` `'>`, `'>`, `">` or `/*>` depending on what MariaDB thinks is needed to complete the command.

`\h` lists the single character commands including `\c` which clears the existing command but, if a quote has not been closed, `\c` needs to be preceded by the appropriate quote to indicate that it is not a continuation of the existing command. The same applies to QUIT; if the quote is not closed, MariaDB will treat QUIT is part of the incomplete command. However, QUIT does not require the ‘`;`’ terminator.

Note that the standard form for connecting to MariaDB is

```
mysql -h host_name -u user_name -p db_name
```

though only ‘`-p`’ is essential once you have secured your MariaDB installation. You can omit `host_name` if you are connecting to *localhost*, `user_name` if the user has the same name as the currently logged in user and `db_name`.

5 Populating the HELP tables

If the command

```
help contents
```

does not display any results, you need to `quit` and enter

```
mysql_install_db --user=root
```

if this is a completely new installation or make sure you are logged in as `root` and enter

```
    source /usr/share/mariadb/fill_help_tables.sql
```

if you have restored a `mysqldump` backup from another system. This will populate the HELP tables.

6 Adding and dropping users

A user with global privileges can add and drop users. The simplest way is to use:

```
    GRANT <privilege list> ON <db_name/tb_name/...>

        TO 'user_name'@'host_name'
        IDENTIFIED BY PASSWORD ’password’;
```

If the user does not exist s/he will be created.

Alternatively, enter

```
    CREATE USER 'user_name'@'host_name'

        IDENTIFIED BY PASSWORD ‘password’;
```

A user created in this way will have no privileges but any user with the GRANT OPTION will be able to grant them privileges. (This is most helpful in a large organisation where a systems administrator with global privileges has to create a new user but the departmental head will need to decide exactly what privileges the user needs in order to do their job.)

Note that ‘password’ will have been stored in your `.mysql_history` file which should be edited to remove all plain text passwords.

If you need to make any parts of your databases accessible to anonymous users, enter

```
    GRANT <privilege list> ON <db_name/tb_name/...>
    TO ' '@'host_name';
```

To drop a user enter

```
    DROP USER 'user_name'@'host_name';
```

but note that this does not take effect until the user’s next session.

7 Granting and revoking privileges

No users apart from `root` should be granted global privileges. Privileges can be granted individually or collectively at one of several levels (table 1 on the next page):

- global
- database
- table
- column
- function
- procedure
- usage, in practice, a user without privileges.

The GRANT OPTION may be granted to a user at any level and they may then grant any privileges they have, including that option, to another user at the same or a lower level.

```
    GRANT <privilege list> ON <db_name/tb_name/...>

        TO 'user_name'@'host_name'
        IDENTIFIED BY PASSWORD ’password’
        WITH GRANT OPTION;
```

Table 1: Some privilege levels

GLOBAL	DATABASE	TABLE	COLUMN	FUNCTION OR PROCEDURE
CREATE USER	CREATE	ALTER	INSERT	ALTER
SHUTDOWN	DROP	CREATE	SELECT	EXECUTE
		DELETE	UPDATE	
		DROP		
		INDEX		
		INSERT		
		SELECT		
		TRIGGER		
		UPDATE		

SHOW DATABASES produces different results depending on the privilege level of the user (table 1).

To grant privileges at column level, specify the columns in a table level GRANT, for example

```
GRANT <privilege> (column3,column5,...)


    [,<privilege> (column4,column6,...)]
    ON <db_name.tb_name>
    TO 'user_name'@'host_name'
    IDENTIFIED BY PASSWORD ’password’;
```

Anyone who has the GRANT OPTION for particular privileges, can revoke those privileges with:

```
REVOKE <privilege list>

    ON <db_name/tb_name/...>
    FROM 'user_name'@'host_name';
```

To revoke the GRANT OPTION enter:

```
REVOKE GRANT OPTION

    FROM 'user_name'@'host_name';
```

Note that the above do not remove the user from the `mysql.user` table; to do that `root` must enter:

```
DROP USER 'user_name'@'host_name';
```

A user with global privileges can revoke all privileges with:

```
REVOKE ALL PRIVILEGES, GRANT OPTION

    FROM 'user_name'@'host_name';
```

8 Changing passwords

A user can change their own password with

```
SET PASSWORD = PASSWORD(‘password’);
```

`root` can change a user’s password with

```
SET PASSWORD FOR 'user_name'@'host_name' =
PASSWORD('password');
```

or

```
USE mysql
UPDATE user SET PASSWORD = PASSWORD('password')

    WHERE host = 'host_name'

        AND user = 'user_name';

FLUSH PRIVILEGES;
```

9 Creating a database

A user with database privileges can create a database with

```
CREATE DATABASE db_name;
```

where `db_name` is the name of the database you wish to create. Such a database takes the default character set and collation; to specify a different character set and collation enter

```
CREATE DATABASE db_name

    CHARACTER SET charset_name
    COLLATE collation_name;
```

While you may use upper and/or lower case characters in database names, you are recommended to use lowercase throughout because Unix database, i.e. directory, names are case sensitive. Using the same name with a different case is permissible in Unix but may create problems if a database is later copied to a Windows or Mac system. Reserved words can be used if surrounded by ` ` but are best avoided.

To use a database, you have to issue the command

```
USE db_name
```

(without the terminating ';') or include it on your initial command line

```
mysql -h host_name -u user_name -p db_name
```

Once you have created several databases, it is advisable to include IF NOT EXISTS to prevent problems arising from creating duplicate databases:

```
CREATE DATABASE IF NOT EXISTS db_name;
```

However, to populate a database with data, you must first create one or more tables.

10 Creating a table

Any user with table privileges for a database can create a table in that database by entering:

```
CREATE TABLE tb_name (col_name1 data_type[(m)],col_name2 data_type[(m)],...);
```

The following column types are available:

Data type	Maximum length	Comments
BIGINT[(m)]		Up to 9223372036854775807 signed; 18446744073709551615 unsigned
BINARY(m)/CHAR BYTE	255 bytes	Binary string type similar to CHAR

Data type	Maximum length	Comments
BIT[(m)]	64 bits	Bit field; default width= 1
BLOB[(m)]	$65,535$ bytes	By default indexed on first 1024 bytes; usage may be limited by packet size and memory
BOOL, BOOLEAN	1	Synonym for TINYINT(1); non-zero values are considered TRUE
CHAR[1]	255 characters	Fixed length string; padded with spaces to the right of data
DATE		YYYY-MM-DD format; 1000-01-01 to 9999-12-31
DATETIME		YYYY-MM-DD HH:MM:SS.000000 format with microsecond precision; 1000-01-01 00:00:00.000000 to 9999-12-31 23:59:59.999999
DECIMAL[(m[,d])]/ DEC[(m[,d])]/ FIXED[(m[,d)]/ NUMERIC[(m[,d])]	65	65 digit decimal number (default 10) with up to 30 decimal places (default 0)
DOUBLE[(m,d)]/ DOUBLE PRECISION [(m),(d)]/ REAL[(m),(d)][2]		A double precision floating point decimal, accurate to about 15 decimal places; the allowable range is hardware dependent
ENUM('`value1`','`value2`', ...)	$65,535$ distinct values	Entries may only be a value, NULL or ' '; the default value is '`value1`' if the field is NOT NULL, otherwise NULL
FLOAT[(m),(d)]		A single precision floating point decimal, accurate to about 7 decimal places; the allowable range is hardware dependent; avoid if possible because all calculations in MariaDB are done with double precision
INT[(m)]/ INTEGER[(m)]		Signed range -2147483648 to 2147483647; unsigned range 0 to 4294967295
LONGBLOB	4GB	By default indexed on first 1024 bytes; usage may be limited by packet size and memory
LONGTEXT	4G characters	The effective maximum length is less if it contains multi-byte values
MEDIUMBLOB	$16,777,215$ bytes	By default indexed on first 1024 bytes; usage may be limited by packet size and memory
MEDIUMINT[(m)]		Signed range -8388608 to 8388607; unsigned range 0 to 16777215

[1]CHAR BYTE = CHAR BINARY

[2]Unless the REAL_AS_FLOAT option is used when it is a synonym for FLOAT

Data type	Maximum length	Comments
MEDIUMTEXT	$16,777,215$ characters	The effective maximum length is less if it contains multi-byte values
SET('`value1`',`value2`, ...)	64 distinct values	A string object with zero or more values from a list of values; returns may be based on a substring which appears in more than one value
SMALLINT[(m)]		Signed range -32768 to 32767; unsigned range 0 to 65535
TEXT	$65,535$ characters	The effective maximum length is less if it contains multi-byte values; where m is specified, the smallest TEXT type large enough to hold m characters is used
TIME		HH:MM:SS format with microsecond precision; -838:59:59.999999 to 838:59:59.999999
TIMESTAMP [(m)]		UNIX timestamp with microsecond precision specified by m (default 0); range 1970-01-01 00:00:01.000000 to 2037-01-09 03:14:07.999999
TINYBLOB	255 bytes	
TINYINT[(m)]		Signed range -128 to 127; unsigned range 0 to 255
TINYTEXT	255 characters	The effective maximum length is less if it contains multi-byte values
VARBINARY(m)	$65,532$ bytes	Binary string type similar to VARCHAR
VARCHAR(m) CHAR VARYING(m) CHARACTER VARYING(m)	$65,532$ bytes	For fields using UTF-8 encoding; trailing spaces retained but entries not padded with spaces. Note that maximum row length is also 65,532 bytes.
YEAR [(2|4)]		Range 1901–2155 in 4 digit format or 1970–2069 in 2 digit format; note that numeric 0000 represents 0000 and 00 represents 2000 whereas "0000" and "00" both represent 2000

The following options are available:

- m number of display digits to 64 (default 10)
- d significant digits/decimal places $0 - 30$
- NOT NULL no NULL values in column
- DEFAULT `<value>` sets a default value for the column
- AUTO_INCREMENT auto-increments the value of an INTEGER type column; only one column in a table may have the AUTO_INCREMENT option
- UNIQUE specifies that all the values in the column must be unique or NULL (unless the NOT NULL option is used)
- UNSIGNED no negative numbers; permits larger numbers to be stored in a column
- ZEROFILL pads out numeric columns with 0s to the left of the stored number; any such column is also UNSIGNED
- SERIAL is an alias for BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE

If an entry which exceeds a column's range is made, it is clipped to the limit of the range.

While you may use upper and/or lower case characters in table names, you are recommended to use lowercase throughout because Unix table, i.e. file, names are case sensitive. Using the same name with a different case is permissible in Unix but may create problems if the files are later copied to a Windows or Mac system. Reserved words can be used if surrounded by ` ` but are best avoided. Column names are not affected by this.

Entries to a date column may be formatted strings, unformatted strings or numbers provided that the data supplied can be fitted to the column; otherwise the column is populated with 0s. Note that entries which appear to be legal may be incorrect.

As with databases, once you have created several tables, it is advisable to include IF NOT EXISTS to prevent problems arising from creating duplicate tables:

```
CREATE TABLE IF NOT EXISTS tb_name (

    col_name1 data_type[(m)],
    col_name2 data_type[(m)],
    ...

);
```

If you wish to create a new empty table with an identical structure to an existing table, you can use

```
CREATE TABLE tb_name2 LIKE tb_name1;
```

If you wish to create a table using data from columns in an existing table, you can enter

```
CREATE TABLE tb_name2 (

    col_name1 data_type[(m)],
    col_name2 data_type[(m)],
    ...

) SELECT <col_name list> FROM tb_name1;
```

11 Adding an index

You can add an index when you create a table with

```
CREATE TABLE tb_name (

    col_name1 data_type[(m)],
    col_name2 data_type[(m)],
    ...

) INDEX (col_name [(m)]);
```

An index is unique to the storage engine being used at the time of its creation; so, if you expect you might to want to drop the index, for example, to use a different storage engine, provide an index name after INDEX. If you do not provide a width value, the whole column will be indexed. If you have used the UNIQUE option on the indexed column, the index can only contain unique values and NULLs.

You can also create an index which uses data from more than one column by specifying a column list rather than a single column. But, where an index covers multiple columns, they must all have the same character set and collation (section 12 on the following page).

If you want to add an index to an existing table, you can either enter:

```
CREATE [UNIQUE] INDEX index_name

    ON tb_name (col_name [(m)]);
```

or

```
ALTER TABLE tb_name
```

```
    ADD [UNIQUE] INDEX (col_name [(m)]);
```

with the same proviso as above, that you need to provide an index name if you expect you might want to drop the index or use a different storage engine.

The column on which you create a UNIQUE index may have NULL as a value but no duplicate values.

To create an index which will be the PRIMARY KEY for the table, use

```
ALTER TABLE tb_name

    ADD PRIMARY KEY (col_name [(m)]);
```

There can only be one PRIMARY KEY per table and it is implicitly NOT NULL. (You cannot use CREATE INDEX for this purpose.)

If you use SELECT to copy column data from another table, specify the indexes before the SELECT statement

```
CREATE TABLE tb_name2 (

    col_name1 data_type[(m)],
    col_name2 data_type[(m)],
    ...

) INDEX (col_name [(m)])


    SELECT <col_name list> FROM tb_name1;
```

11.1 FULLTEXT indexes

The MyISAM storage engine provides the option of creating a FULLTEXT index on CHAR, VARCHAR and TEXT columns either on creation of the table with

```
CREATE TABLE tb_name (

    col_name1 data_type[(m)],
    col_name2 data_type[(m)],
    ...

) FULLTEXT (col_name);
```

or by using ALTER TABLE or CREATE INDEX.

```
ALTER TABLE tb_name

    ADD FULLTEXT (col_name);


CREATE FULLTEXT INDEX index_name

    ON tb_name (col_name);
```

12 Character sets and collations

A table will have the character set and collation of the database to which it belongs; you can change this from the outset with:

```
CREATE TABLE tb_name <col_name list>

    CHARACTER SET charset_name
    COLLATE collation_name;
```

or alter it with

```
ALTER TABLE tb_name

    CHARACTER SET charset_name
    COLLATE collation_name;
```

It is also possible to specify a different character set or collation for a single column provided it is VARCHAR, CHAR or TEXT with

```
CREATE TABLE tb_name (
        col_name1 data_type[(m)],
        col_name2 data_type[(m)]
            CHARACTER SET charset_name
            COLLATE collation_name,
        col_name3 data_type[(m)],
        ...
    );
```

To specify `utf8` for a CHAR or VARCHAR column add NATIONAL before the column specification:

```
    CREATE TABLE tb_name (
        col_name1 data_type[(m)],
        NATIONAL col_name2 [CHAR | VARCHAR][(m)],
        col_name3 data_type[(m)],
        ...
    );
```

MariaDB stores its own data, for example, database, table and column names, in `utf8` but the default character set is `latin1` and the default collation `latin1_swedish_ci`, that is, case insensitive Swedish which happens to work well for most Western European languages.

A collation determines how data is compared or sorted; collations in `.ci` are case insensitive, those in `.cs` are case sensitive and those in `.bin` are binary collations. They only becomes significant if you communicate information such as table or column names to another server which needs to know the character set of this data as well as the data from the tables.

The default character set and collation are used where no character set or collation is specified at database, table or, in the case of CHAR, VARCHAR and TEXT columns, column level.

By using the COLLATE qualifier, it is possible to display results in a different collation from the one in which they are stored, for example,

```
    SELECT * FROM TABLE tb_name
        WHERE <condition>
        ORDER BY col_name
        COLLATE collation_name;
```

To compare strings from different character sets use

```
    CAST(<expression>
        AS data_type
        CHARACTER SET charset_name
    )
```

in an appropriate statement. To transfer a string from one character set in one column to a column using a different character set

```
    INSERT INTO tb_name(col_name)

        SELECT CONVERT(col_name USING charset_name)
        FROM tb_name [WHERE <condition>];
```

Occasionally the use of different collations in the same statement throws up unexpected results. See Character sets and collations for further information.

13 Nomenclature

Normally you can refer to a database, table or column by its name alone but where tables with the same name from different databases or columns with the same name from different tables are being used, you may have to prepend the database or table name or both:

```
db_name.tb_name
```

or

```
tb_name.col_name
```

or

```
db_name.tb_name.col_name
```

Since database and table names conform to the style of the operating system, they are case sensitive in Unix but not in Windows; however, while column names are not case sensitive, case should be consistent within a query.

Note that InnoDB tables use lower case throughout (so all existing table names should be converted to lowercase before using InnoDB) and that when using ODBC it may be necessary to use

```
.table
```

to refer to a table in the current database.

14 Altering a table

You can alter a table in a variety of ways including

```
ALTER TABLE tb_name

    ADD col_name data_type[(m)]
    [FIRST | AFTER col_name];


ALTER TABLE tb_name


    ADD [UNIQUE] INDEX (col_name [(m)]);


ALTER TABLE tb_name

    ADD PRIMARY KEY (col_name [(m)]);


ALTER TABLE tb_name

    ADD FULLTEXT (col_name);


ALTER TABLE tb_name


    CHANGE old_col_name new_col_name data_type[(m)]
    [FIRST | AFTER col_name];


ALTER TABLE tb_name

    MODIFY col_name data_type[(m)]
    [FIRST | AFTER col_name];


ALTER TABLE tb_name

    DROP col_name;


ALTER TABLE tb_name

    DROP INDEX index_name;


ALTER TABLE tb_name

    ORDER BY col_name[, col_name,...];


ALTER TABLE tb_name
```

```
        RENAME new_tb_name;
```

Note that adding a PRIMARY KEY to a column all of whose values are not UNIQUE will generate an error

15 Inserting data

To enter a single row of data you can use

```
    INSERT INTO tb_name

        VALUES(value1,value2,...);
```

In this case there should be one value for each column in the table; if a column is empty, its `value` must be NULL (without inverted commas).

To enter multiple rows of data you can add further VALUES expressions:

```
    INSERT INTO tb_name

        VALUES(value1,value2,...),
        VALUES(value1,value2,...)
        ...;
```

If most of the columns are to take a DEFAULT value, you can use

```
    INSERT INTO tb_name (col_name3,col_name7,...)

        VALUES (value3,value7,...);
```

which will give unnamed columns their DEFAULT value. Alternatively, you can use

```
    INSERT INTO table


        SET col_name3=value3,col_name7=value7,...;
```

where DEFAULT is an acceptable value.

You can also copy values from another table by using

```
    INSERT INTO tb_name2

        (col_name1,col_name2,...)
        SELECT tb_name1.col_name1,col_name2,...
        FROM tb_name1 WHERE <condition>;
```

Omit the `WHERE <condition>` to copy all the rows from `tb_name1` into `tb_name2`.

If the table has a UNIQUE or PRIMARY KEY column, use REPLACE rather than INSERT

```
    REPLACE INTO tb_name

        VALUES(value1,value2,...);


    REPLACE INTO tb_name

        (col_name1,col_name2,...)
        VALUES (value1,value2,...);


    REPLACE INTO tb_name


        SET col_name1=value1,col_name2=value2,...;


    REPLACE INTO tb_name2

        (col_name1,col_name2,...)
        SELECT tb_name1.col_name1,col_name2,...
        FROM tb_name1 WHERE <condition>;
```

As this deletes any row in the table (`tb_name2`) which has the same value for a UNIQUE or PRIMARY KEY column as the row being SELECTed from the other table (`tb_name1`) and inserts the row being SELECTed, be clear that this is what you want to happen.

16 Importing data from external sources

To import a text file you can use

```
    LOAD DATA INFILE 'file_name'

        INTO TABLE tb_name;
```

where ‘`file_name`’ is the full path and filename. Empty columns must be filled with `\N`.

The default format for data in text files is columns separated by tabs and rows terminated by line feeds, as if you had added

```
    FIELDS TERMINATED BY '\t'

        ENCLOSED BY ' '
        ESCAPED BY '\\'
        LINES STARTING BY ' '
        TERMINATED BY '\n'
```

To specify a different separator use FIELDS TERMINATED BY '`<separator>`' or [OPTIONALLY] ENCLOSED BY '`<character>`'; to specify a different terminator, use LINES TERMINATED BY '`<terminator>`'. For example, to import a CSV file from DOS/Windows use

```
    LOAD DATA INFILE 'file_name'

        INTO TABLE tb_name
        FIELDS TERMINATED BY ','
        LINES TERMINATED BY '\r\n';
```

If the external data file does not contain all the columns of the MariaDB table, you can add a column list to the command

```
    LOAD DATA INFILE 'file_name'

        INTO TABLE tb_name(
           col_name1,col_name2,...
        );
```

You can also cause MariaDB to ignore a column in the data file by assigning it to a user variable which is then discarded

```
    LOAD DATA INFILE 'file_name'

        INTO TABLE tb_name (

           col_name1,@dummy,col_name2,...
        );
```

Finally, you can include a value for a column which is not in the external data file while loading data by using SET with a column list

```
    LOAD DATA INFILE 'file_name'

        INTO TABLE tb_name(
           col_name1,col_name2,...
        )
        SET columnx=valuex;
```

17 Using dates and times

DATE, DATETIME and TIME represent the whole or part of a date and/or time from 1000-01-01 00:00:00.000000 to 9999-12-31 23:59:59.999999. YEAR can be used for dates between 1901 and 2155 while TIMESTAMP can be used for dates and times from 1970-01-01 00:00:00.000000 to 2037-01-09 03:14:07.999999. Alternative delimters can be used, e.g. 2006/04/14 10.56.34 as long as the pattern makes sense while strings or numbers without delimiters which make a sensible date and time will be stored with the standard delimiters. However, TIME values without delimiters will be considered elapsed time; TIME values delimited with ‘:’ will be considered time of day.

Also, as long as delimiters are used, forms such as '99-1-4 6:10:30' will be stored correctly, in this case as '1999-01-04 06:10:30'. Invalid sequences with or without delimiters are stored as all or part of '0000-00-00 00:00:00'.

It is more efficient to store a NULL or empty field as all or part of '0000-00-00 00:00:00'; in the case of TIMESTAMP, use '00000000000000'. Both can be entered as '0'.

To allow calculations of time over more than a day, TIME can also be expressed as 'D 00:00:00' where 'D' is a number of days.

By default, the first TIMESTAMP column in a table is set to

```
DEFAULT CURRENT_TIMESTAMP
ON UPDATE CURRENT_TIMESTAMP
```

When a date or time expression is used in a numeric expression it takes the form YYYYMMDDHHMMSSmmmmmm. To ensure a numeric representation, add +0 to the date or time expression.

18 Retrieving data

When retrieving data, note that MariaDB returns NULL rather than '0' for false or '1' for true if the values being compared include NULL; the operators IS NULL and IS NOT NULL may help to identify these situations.

18.1 Retrieving data using SELECT

Data can be displayed on screen with

```
SELECT col_name1,col_name2,...

    FROM tb_name WHERE <condition>;
```

The column list can be replaced by the wildcard '`*`' if all columns are required. Rows can only be selected using `WHERE <condition>`. Omitting the `WHERE <condition>` results in all the rows in the table being displayed. The `<condition>` can contain AND or OR with brackets being used to indicate precedence.

This displays results horizontally on screen; to display the results vertically use `\G`

```
SELECT col_name1,col_name2,...

    FROM tb_name WHERE <condition>\G
```

To display more than one set of results, separate the queries with ',' or ';'

```
SELECT VERSION(), CURRENT_DATE;
SELECT col_name1 FROM tb_name1;
SELECT col_name2 FROM tb_name2;
```

Duplicate entries in a column can be suppressed with DISTINCT

```
SELECT DISTINCT col_name FROM tb_name;
```

18.1.1 Retrieving data from more than one table

To retrieve data from more than one table, specify the columns as `tb_name.col_name` where both tables contain columns with the same name as well as both tables:

```
SELECT tb_name1.col_name1,col_name2,...

    FROM tb_name1, tb_name2
    WHERE tb_name1.col_name = tb_name2.col_name;
```

To join the tables for the search use [LEFT] JOIN and ON or USING instead of WHERE

```
SELECT col_name1,col_name2,... FROM tb_name1

    LEFT JOIN tb_name2 ON <expression>;
```

```
SELECT col_name1,col_name2,... FROM tb_name1

    LEFT JOIN tb_name1 USING (col_name1,col_name2,);
```

18.1.2 Retrieving data using comparisons

Comparisons are made using LIKE and NOT LIKE; the wildcards _ and % (use % for 0 characters) can come anywhere in the expression; to extract everyone whose last name begins with ‘B’ enter

```
SELECT * FROM tb_name

    WHERE lastname LIKE 'b%';
```

To extract everyone whose last name ends with ‘son’ enter

```
SELECT * FROM tb_name

    WHERE lastname LIKE '%son';
```

To extract everyone whose last name contains ‘v’ enter

```
SELECT * FROM tb_name

    WHERE lastname LIKE '%v%';
```

Note that each of these comparisons are case insensitive.

Each of the above can be extracted using REGEXP which compares only part of the string; to extract everyone whose last name begins with ‘B’ enter

```
SELECT * FROM tb_name

    WHERE lastname REGEXP '^b';
```

If you want a case sensitive match use BINARY:

```
SELECT * FROM tb_name

    WHERE lastname REGEXP BINARY '^b';
```

To extract everyone whose last name ends with ‘son’ enter

```
SELECT * FROM tb_name

    WHERE lastname REGEXP 'son$';
```

To extract everyone whose last name contains ‘v’ enter

```
SELECT * FROM tb_name

    WHERE lastname REGEXP 'v';
```

To extract every instance of where someone’s last name begins with ‘A,’ ‘B’ or ‘C’ enter

```
SELECT * FROM tb_name

    WHERE lastname REGEXP '^[abc]';
```

To extract every instance where someone’s last name ends with ‘ll’ enter

```
SELECT * FROM tb_name

    WHERE Name REGEXP 'l*$';
```

Note that RLIKE and NOT RLIKE are synonyms of REGEXP and NOT REGEXP.

To select last names which are six characters long enter either

```
SELECT * FROM tb_name

    WHERE lastname LIKE '______';
```

or

```
SELECT * FROM tb_name

    WHERE lastname REGEXP '^......$';
```

or

```
SELECT * FROM tb_name
    WHERE lastname REGEXP '^.{6}$';
```

using the repeat *n* times operator.

18.1.3 Retrieving different data from the same table for comparison

Sometimes it is useful to treat data within a table as coming from a different table:

```
SELECT p1.col_name1, p1.col_name2,
p2.col_name1, p2.col_name4
    FROM tb_name AS p1, tb_name AS p2
    WHERE p1.col_name1 = p2.col_name1
```

Here `col_name1` will be displayed twice and `col_name2` and `col_name4` once each.

18.1.4 Retrieving data using CASE and IF

Data can be retrieved using CASE statements to make a comparison or as a conditional.

```
SELECT CASE value1 WHEN value2
    THEN result [WHEN value3
       THEN result]
[ELSE result]
END;
```

In this case `value1` is compared with each successive value and if the comparison is valid the result is acted on. In the next case each condition is evaluated and the result of the first successful evaluation is acted upon.

```
SELECT CASE WHEN condition1 THEN result
    WHEN condition2 THEN result
ELSE result
END;
```

Where there are only two cases, use IF

```
SELECT IF(expr1,expr2,expr3);
```

Note that, if `expr1` is numeric, it must be an integer. You can get round this by comparing a floating point number with 0 (`expr1<>0,...`) since this will return 1 and lead to `expr2` being acted on.

18.1.5 Retrieving data using GROUP BY

You can obtain the AVERAGE of the entries in one column grouped by the identical entries in another with

```
SELECT col_name1, AVG(col_name2)
    FROM tb_name GROUP BY col_name1;
```

Functions such as COUNT(), MAX(), MIN(), SUM() and the various statistical functions can be used in a similar way. To see the results from each group aggregated use WITH ROLLUP:

```
SELECT col_name1, SUM(col_name2)
    FROM tb_name GROUP BY col_name1
    WITH ROLLUP;
```

Note that ORDER BY (section 18.1.6 on the next page) cannot be used in the same statement as WITH ROLLUP.

To select only the values that match a particular criterion, using HAVING `<condition>`

```
SELECT col_name1, AVG(col_name2)
```

```
    FROM tb_name GROUP BY col_name1
    HAVING AVG(col_name2)>20;
```

To see all the values in a particular group use

```
SELECT col_name1, GROUP_CONCAT(col_name2)

    FROM tb_name GROUP BY col_name1;
```

To see them in ascending order with a separator between each value use

```
SELECT col_name1, GROUP_CONCAT(


    col_name2 ORDER BY col_name3 SEPARATOR ' '
    ) FROM tb_name GROUP BY col_name1;
```

18.1.6 Retrieving data using ORDER BY

To see a display sorted on a column, add 'ORDER BY col_name'

```
SELECT col_name1,col_name2,...

    FROM tb_name [WHERE <condition>]
    ORDER BY col_name3;
```

This will be case insensitive — 'MacDonald' may come before or after 'Macdonald'; to make the sort case sensitive add BINARY before the column name

```
SELECT col_name1,col_name2,...

    FROM tb_name [WHERE <condition>]
    ORDER BY BINARY col_name3;
```

To reverse the sort order, add DESC after the column name

```
SELECT col_name1,col_name2,...

    FROM tb_name [WHERE <condition>]
    ORDER BY [BINARY] col_name3 DESC;
```

To sort on multiple columns, enter a list of column names

```
SELECT col_name1, col_name2,...

    FROM tb_name [WHERE <condition>]
    ORDER BY BINARY col_name3, col_name4 DESC;
```

In this case BINARY applies to the first column to be sorted and DESC to the second column to be sorted.

You can limit the number of rows being returned with LIMIT n which returns the first n rows that match the SELECT expression; LIMIT o,n returns the first n rows after o rows. To select the first 20 matching rows, use

```
SELECT col_name1,col_name2,...

    FROM tb_name [WHERE <condition>]
    ORDER BY col_name3 LIMIT 20;
```

To select a random sample use ORDER BY RAND() LIMIT n where n is the sample size required:

```
SELECT *

    FROM tb_name [WHERE <condition>]
    ORDER BY RAND() LIMIT n;
```

18.1.7 Retrieving data using BOOLEAN MODE

You can refine certain searches by using BOOLEAN MODE:

```
SELECT * FROM tb_name

    WHERE MATCH (col_name1,col_name2)
    AGAINST ('str' IN BOOLEAN MODE);
```

You can bring even more precision by using + and - to indicate required and excluded strings:

```
SELECT * FROM tb_name

    WHERE MATCH (col_name1,col_name2)
    AGAINST ('+str1 -str2' IN BOOLEAN MODE);
```

Note that ‘apple banana’ finds either ‘apple’ or ‘banana,’ ‘+apple banana’ rates rows with ‘apple’ in them higher, whereas ‘+apple +banana’ only finds rows that contain both ‘apple’ and ‘banana.’ You also use the wildcard * at the end of a word, < and > to indicate less or more importance in the search, ~ to mark noise words that you want ignored (rather than the whole row in which they occur), “ ” to indicate an exact match and ( ) to indicate precedence.

18.1.8 Retrieving the results of a calculation

To display the results of a calculation using data retrieved from a table, use AS to create an alias for a temporary column

```
SELECT name,birth,(

    YEAR(CURRENT_DATE()-YEAR(birth)

)-(


    RIGHT(CURRENT_DATE(),5)<RIGHT(birth,5)

)AS age FROM tb_name;
```

This displays a list of names, dates of birth and ages derived from dates of birth.

To use the same columns twice in a query, give each column a table alias:

```
SELECT

    alias1.col_name1,alias1.col_name2,
    alias2.col_name1,alias2.col_name2
    FROM tb_name AS alias1, tb_name AS alias2
    WHERE <condition>;
```

To retrieve data using two SELECT statements and a UNION, place the SELECT statements in parentheses; for example

```
(SELECT col_name1,col_name2,...

    FROM tb_name WHERE <condition>)

UNION
(SELECT col_name3,col_name4,...

    FROM tb_name WHERE <condition>)

ORDER BY col_name5;
```

To display a person’s age using a date of birth column, enter

```
SELECT name, dob, (

    YEAR(
       CURDATE()) - YEAR(dob)
    ) - (


       RIGHT(CURDATE(),5)<RIGHT(dob,5)
    )

AS age FROM tb_name;
```

The condition (RIGHT ... (Dob,5)) tests whether the birthday has passed in the current year and deducts a year if it has not.

If any of the cells in a column are likely to hold NULL values, add the condition ‘WHERE column IS NOT NULL’ — for example:

```
WHERE dob IS NOT NULL
```

18.1.9 Retrieving data using date functions

Apart from retrieving data using the standard date functions such as DAYOFMONTH(), MONTH() and YEAR(), it is also possible to retrieve data for periods in the future using the DATE_ADD() function. To find out whose birthdays are coming up next month enter

```
SELECT name, birth

    FROM tb_name
    WHERE MONTH(birth) =  MONTH(

       DATE_ADD(CURDATE(), INTERVAL 1 MONTH)
    );
```

18.1.10 Retrieving data using other criteria

To retrieve data using criteria such as maximum (or minimum) values enter

```
SELECT MAX(col_name)

    AS col_name FROM tb_name;
```

To retrieve the whole row in which this data occurs enter

```
SELECT * FROM tb_name


    WHERE col_name=(SELECT MAX(col_name)
    FROM tb_name);
```

To retrieve the maximum value in each of a group of entries enter

```
SELECT col_name1, MAX(col_name2)

    AS col_name2
    FROM tb_name
    GROUP BY col_name1;
```

To retrieve the rows in which this data occurs enter

```
SELECT col_name1, col_name2, col_name3

    FROM tb_name s1
    WHERE col_name3 = (

       SELECT MAX(s2.col_name3)

    FROM tb_name s2
    WHERE s1.col_name1=s2.col_name1;
```

18.1.11 Counting while retrieving data

To count the number of instances in a column of a MyISAM table, use COUNT(*):

```
SELECT COUNT(*) FROM tb_name;
```

to return the number of rows in a table. However, most columns will include duplicate entries; so GROUP BY must normally be used to avoid an error message:

```
SELECT col_name,

    COUNT(*) FROM tb_name
    GROUP BY col_name;
```

More than one column may be specified by GROUP BY and the number of rows restricted by the WHERE <condition>

```
SELECT col_name, col_name,

    COUNT(*) FROM tb_name
    WHERE <condition>
    GROUP BY col_name, col_name;
```

18.2 Retrieving data using FULLTEXT searches

FULLTEXT searches are only available with MyISAM tables. Before retrieving data using FULLTEXT searches, it is desirable, and may be necessary, to create the appropriate FULLTEXT indexes (Section 11.1). To search for a text string of more than four characters in a CHAR, VARCHAR or TEXT column, enter

```
SELECT * FROM tb_name

    WHERE MATCH (col_name1,col_name2)
    AGAINST ('str');
```

Note that

- multiple columns must be in the same FULLTEXT Index
- MariaDB ignores
  - a standard list of very common words and
  - any string that occurs in more than 50% of the rows.

You can avoid or change the former by adding

```
[mysqld]
ft_stopword_file=/etc/mysql/stopword.txt
```

to `/etc/my.cnf` where `stopword.txt` is an empty file to ensure every word is indexed or a different list of words to be ignored from those in the default list.

To extend a FULLTEXT search for something which might have synonyms use:

```
SELECT * FROM tb_name

    WHERE MATCH (col_name1,col_name2)
    AGAINST ('str' WITH QUERY EXPANSION);
```

The results of the highest rated results are used to undertake a second search for rows which may include the same concepts but not the search string.

To adjust the relevance of a FULLTEXT search use:

```
SELECT * FROM tb_name

    WHERE MATCH (col_name1,col_name2)
    AGAINST ('str' IN BOOLEAN MODE);
```

and prepend each word in `str` with the appropriate BOOLEAN MODE operator (table 4 on page 47)

To discover the relevance of each row in the response, use

```
SELECT col_name1,

    MATCH (col_name1,col_name2)
    AGAINST ('str')
    FROM tb_name;
```

To obtain an ordered list use

```
SELECT col_name1,

    MATCH (col_name1,col_name2)
    AGAINST ('str') AS score
    FROM tb_name

        WHERE MATCH (col_name1,col_name2)
        AGAINST ('str')
        FROM tb_name;
```

18.3 Retrieving data using HANDLER

HANDLER provides an easier way of retrieving data when using an interactive user interface or prepared statements

```
HANDLER tb_name OPEN;
```

opens a table;

```
HANDLER tb_name READ index WHERE <condition>;
```

displays the first matching row; use LIMIT n to display more rows. You can then move around the database by using FIRST, LAST, NEXT or PREV instead of the WHERE condition

```
HANDLER tb_name READ index NEXT;
```

reads the next row in the table. If you choose not to use an index, only FIRST and NEXT are available.

```
HANDLER tb_name CLOSE;
```

closes a table.

19 Prepared statements

If you are going to execute the same complex statement many times within a single session, you can create a prepared statement with:

```
PREPARE stmt_name FROM "<statement>";
```

and then execute it whenever you need it with

```
EXECUTE stmt_name;
```

To use the same prepared statement with different tables, use the HANDLER commands.

If you expect to use a prepared statement again another day, save it as a stored routine.

20 Stored routines

MariaDB offers a variety of stored routines:

- functions
- procedures
- triggers
- events
- views

20.1 Stored functions

A stored function normally consists of a name, a number of arguments, a series of instructions on how to manipulate the arguments and an instruction on how to return the result when called by SELECT statement such as:

```
select function_name();
```

20.2 Stored procedures

A stored procedure contains statements which will be executed when the procedure in called with:

```
call procedure_name();
```

20.3 Triggers

Triggers are procedures that only execute after a particular event associated with a particular table.

20.4 Events

Events are procedures that execute at some later stage, whether once off or at regular intervals, associated with a particular database.

20.5 Views

Views are procedures that return the results of a query, normally enabling a complex query to be simplified.

For more information on stored procedures, see https://mariadb.com/kb/en/stored-programs-and-views/.

21 Dynamic columns

This feature was added with MariaDB 5.3. For more details see Dynamic Columns Functions.

22 Changing data

To change data in an existing table use

```
UPDATE tb_name
    SET col_name3 = 'value',
    SET col_name7 = 'value',
    ...
    WHERE <condition>;
```

If you have set `safe-updates` (`-U`), MariaDB will ignore an update without a WHERE or a LIMIT condition; if not, without a WHERE or LIMIT condition, it will update every row in the column.

23 Deleting data

To delete rows from a table enter

```
DELETE FROM tb_name
    WHERE <condition>;
```

Omitting the '`WHERE <condition>`' results in the whole table being deleted unless you have set safe updates (`-U`) in which case any command without a WHERE or LIMIT condition will be ignored. Using ORDER BY with LIMIT allows a particular group of rows to be selected for deletion.

Where a number of rows in an indexed table are to be deleted, it may be quicker to use

```
DELETE QUICK FROM tb_name
    WHERE <condition>;
```

which does not update the indices, followed by

```
OPTIMISE TABLE tb_name;
```

It is also possible to delete rows from more than one table at once using

```
DELETE tb_name1[, tb_name2,...]
    FROM <table references>
    WHERE <condition>;
```

24 Saving retrieved data to file

To save retrieved data to file from a single search use

```
SELECT col_name1,col_name2,...
    INTO OUTFILE 'file_name'
    FROM tb_name WHERE <condition>;
```

To save retrieved data to file continuously use

```
tee file_name;
```

or

```
\T file_name;
```

To stop data being saved to the file use

```
notee;
```

or

```
\t;
```

If, however, the data is binary, such as that contained in a BLOB, you need to use:

```
SELECT col_name7
INTO DUMPFILE 'file_name'
    FROM tb_name WHERE <condition>;
```

As there is no sense in putting two binary strings into a file, you will normally get an error if you try to do this.

25 Backing up and restoring databases

To back up all your databases, `quit` MariaDB and enter:

```
mysqldump -u root -p --all-databases > /path/to/filename.sql
```

To restore all your databases, enter

```
mysql -u root
source /path/to/filename.sql
flush privileges;
```

if you are using data from an earlier `mysqldump` backup whether of MySQL or MariaDB databases, or

```
mysql -u root p
source /path/to/filename.sql
flush privileges;
```

if you need to restore a backup of your existing databases.

Since `mysqldump` stores and reads standard SQL, you can use it for transferring data between different database programs as long as there are no program specific features in the databases.

26 Comments

There are three supported comment styles in MariaDB:

- From a '`#`' to the end of a line:

  ```
  SELECT * FROM users; # This is a comment
  ```

- From a '`--`' to the end of a line. The space after the two dashes is required

  ```
  SELECT * FROM users; -- This is a comment
  ```

- C style comments from an opening '`/*`' to a closing '`*/`.' Comments of this form can span multiple lines:

  ```
  SELECT * FROM users; /* This is a
  multi-line
  comment */
  ```

27 Creating user variables

A user variable can be created with

```
SET @variable1=value1[, @variable2=value2,...];
```

It can be used in statements such as

```
SELECT @variable:=value FROM tb_name;
```

Note the use of `:=` rather than `=` in statements outside `SET @user_variable`. Note also that, while it is possible to create a user variable within a SELECT statement, you should not use a `user_variable` in a HAVING, GROUP BY or ORDER BY statement which has not already been specified in a previous SET or SELECT statement.

28 Batch processing

MariaDB can process batch files; to execute a batchfile from outside MariaDB enter

```
mysql -h host_name -u user_name -p < batchfile
```

To save the output of the batchfile on screen or to file, enter

```
mysql -h host_name -u user_name -p < batchfile|more
mysql -h host_name -u user_name -p < batchfile>mysql.out
```

To execute a batchfile from within MariaDB enter

```
SOURCE batchfile;
```

or

```
\. batchfile
```

29 Error codes

If you want to find out what a MariaDB error code means, quit MariaDB and enter

```
perror n
```

where n is the number of the error code.

30 Configuring MariaDB

Many aspects of MariaDB can be configured by changing the settings in `/etc/my.cnf`; if the installed version (actually `/usr/share/mysql/my-medium.cnf`) does not suit your needs, you can swap it for any of:

```
/usr/share/mysql/my-huge.cnf
/usr/share/mysql/my-innodb-heavy-4G.cnf
/usr/share/mysql/my-large.cnf
/usr/share/mysql/my-medium.cnf
/usr/share/mysql/my-small.cnf
```

As `root`, you can find out the current status of any aspect of a MariaDB installation by entering

```
use information_schema
show tables;
```

and then selecting the table which holds the information about the aspect of MariaDB in which you are interested.

MariaDB supports a variety of storage engines, of which MyISAM is normally the default unless the server is configured for transactions in which case the XtraDB engine will normally be configured; this is a replacement for the InnoDB engine and so all the commands associated with it begin `inno*`.

To see which engines are installed, enter

```
show engines;
```

To associate a particular table with a particular engine, enter

```
engine engine_name
```

at the end of the table creation definition.

A Commands

A.1 Short forms

\? = HELP

\# = REHASH

\. = SOURCE

\s = STATUS

\! = SYSTEM

\C CHARSET

\c CLEAR existing command (after terminating any outstanding string with the appropriate quote; otherwise '\c' will be treated as part of the string and not as a command)

\d = DELIMITER

\e = EDIT

\G = EGO

\g = GO

\h = HELP

\n = NOPAGER

\P = PAGER.

\p = PRINT

\q = QUIT

\R = PROMPT

\r = CONNECT

\t = NOTEE

\T = TEE

\u = USE

\W = WARNINGS

\w = NOWARNING

user_var_name:=value create a user variable whose value is `value`

A.2 Long forms[3]

ALTER DATABASE db_name alter the character set or collation of a database

CHARACTER_SET charset_name change character set

COLLATE collation_name change collation

ALTER EVENT event name alter an existing event

ALTER FUNCTION function name alter an existing function

ALTER PROCEDURE procedure name alter an existing procedure

ALTER TABLE tb_name

ADD col_name data_type[(m)] add a new column at the end of the row; use `FIRST` or `AFTER col_name` to insert a column elsewhere in the table

ADD FULLTEXT [index name] (col_name) add a full-text index for column, using the column name if no index name supplied

ADD INDEX [index name] (col_name) add an index for column, using the column name if no index name supplied

[3] *db_name.tb_name* can be used where there might be ambiguity about the table; string values should be enclosed in ' '; most long forms should be terminated with ; or \G

- **ADD PRIMARY KEY (col_name)** add a primary key column
- **ADD UNIQUE [index name] (col_name)** add a unique index for column, using the column name if no index name supplied
- **CHANGE col_name1 col_name2 data_type[(m)]** change the specification of `col_name1`; `col_name2` may be the same as `col_name1` if the column name is not to be changed
- **CHARACTER SET charset_name;** change the default character set for the table
- **CONVERT TO CHARACTER SET charset_name;** change the character set for all existing CHAR, VARCHAR and TEXT columns
- **DISABLE KEYS** turn off key update
- **DROP col_name** delete a column
- **DROP index_name** delete an index
- **DROP PRIMARY KEY** delete a primary key
- **ENABLE KEYS** turn on key update
- **MODIFY col_name data_type[(m)]** modify column; use `FIRST` or `AFTER col_name` to move a column elsewhere in the table
- **ORDER BY col_name** alter the order of rows in a table
- **RENAME [TO] new name** rename table

ALTER VIEW view_name alter an existing view

ANALYZE TABLE tb_name; analyses a MyISAM, Aria or InnoDB table, particularly with reference to index use; returns 1 on a unique column suggesting it is suitable for a unique index

CHECK TABLE tb_name; checks an Archive, Aria, CSV, InnoDB or MyISAM

CHECKSUM TABLE tb_name; checks or calculates the checksum for an Aria or MyISAM table, useful for discovering if two tables are identical

CREATE DATABASE [IF NOT EXISTS] db_name;

- **CHARACTER SET charset_name** set the default character set for the database
- **COLLATE collation_name** set the default collation for the database

CREATE EVENT event_name create an event

CREATE FUNCTION function name create a function

CREATE INDEX index name ON tb_name (col_name(n)); create an index using the first n characters of column

CREATE PROCEDURE procedure name create a procedure

CREATE TABLE tb_name (col_name (data_type(n))[, col_name(data_type(n)),. . .]); create a table with the columns specified in the column list

CREATE TABLE tb_name1 SELECT * FROM tb_name2; create a new table using the columns in `tb_name2`

CREATE TRIGGER trigger name create a trigger

CREATE VIEW view_name create a view

DELETE FROM tb_name WHERE <condition>; delete specified rows in a table; omitting the '`WHERE <condition>`' deletes all rows in the table

DELIMITER set statement delimiter; takes the rest of the line as a new delimiter; used with stored routines to enable ';' to be used as the delimiter between multiple statements in a stored routine without triggering an end of statement

DESCRIBE tb_name; display table structure

DROP DATABASE db_name; delete database

DROP EVENT event_name; delete event

DROP FUNCTION function_name; delete function

DROP INDEX index ON tb_name; delete index

DROP PREPARE stmt_name; delete statement

DROP PROCEDURE procedure name; delete procedure

DROP TABLE tb_name; delete table

DROP TRIGGER trigger_name delete trigger

DROP VIEW view_name delete view

DROP USER 'user_name'@'host_name'; delete user

EXECUTE stmt_name execute a prepared statement

EXIT = QUIT

EXPLAIN tb_name; =DESCRIBE tb_name

EXPLAIN [EXTENDED] SELECT...; explain how MariaDB processed the SELECT query

FLUSH PRIVILEGES; update privilege tables

FLUSH QUERY CACHE; optimise the query cache

FLUSH STATUS; reset status variables to 0

FLUSH TABLE; ensure that all pending changes are written and resets the query cache

GRANT <privilege_list> ON <db_name/tb_name/...> TO 'user_name'@'host_name'; grant privileges to a user at a specified level; if a new user, add `IDENTIFIED BY 'password'`

HANDLER tb_name action perform the actions OPEN, READ or CLOSE on a table with qualifiers, WHERE and LIMIT conditions allowing prepared statements rather than SELECT queries to be executed

HELP expr display the help available for `expr`

HELP CONTENTS display the top level list of help topics

INSERT INTO tb_name [(col_name1,col_name2,...)] VALUES (value1,value2,...); insert a row into a table without a primary key or unique index (otherwise use REPLACE) with the following values; if an incomplete list of values is given, those columns which are not specified in the column list must have a default value; if a `value` to be inserted is NULL, use NULL without inverted commas

INSERT INTO tb_name SET col_name1=value1, col_name2=value2,...; insert a row into a table without a primary key or unique index (otherwise use REPLACE) giving the specified columns these values; if an incomplete list of columns is given, those columns which are not specified in the column list must have a default value; if a `value` to be inserted is NULL, use NULL without inverted commas

INSERT INTO tb_name [(col_name1,col_name2,...)] SELECT...; insert rows from another table into a table without a primary key or unique index (otherwise use REPLACE); do not use DELAYED; if an incomplete list of columns is given, those columns which are not specified must have a default value

LOAD DATA INFILE 'file_name' INTO TABLE tb_name; insert records from an external data file; qualifiers defining delimiters and terminators and an optional column list may be added if the external data file uses non-standard delimiters or terminators or does not contain all the columns in the table

OPTIMIZE TABLE tb_name; tidy up deleted records, etc.

PREPARE stmt_name FROM statement prepare a statement for later execution

QUIT terminate MariaDB session; does not require ';' terminator

RENAME TABLE tb_name1 TO tb_name2 [, tb_name3 TO tb_name4,...] rename table

RENAME USER user1 TO user2[, user3 TO user4,...] rename user

REPAIR TABLE tb_name; repair an Archive, Aria, CSV or MyISAM table

REPLACE INTO tb_name [(col_name1,col_name2,. . .)] VALUES (value1,value2,. . .); insert a row in a table with a primary key or unique index (otherwise use INSERT) with the following values; if an incomplete list of values is given, those columns which are not specified in the column list must have a default value; if a `value` to be inserted is NULL, use NULL without inverted commas; if a value to be inserted in a primary key or unique index is the same as an existing one, the existing row is deleted and the new row inserted

REPLACE INTO tb_name SET col_name1=value1, col_name2=value2,. . . ; insert a row in a table with a primary key or unique index (otherwise use INSERT) giving the specified columns these values; if an incomplete list of values is given, those columns which are not specified in the column list must have a default value; if a `value` to be inserted is NULL, use NULL without inverted commas; if a value to be inserted in a primary key or unique index is the same as an existing one, the existing row is deleted and the new row inserted

REPLACE INTO tb_name SELECT. . . ; insert rows from another table in a table with a primary key or unique index (otherwise use INSERT); do not use DELAYED; if an incomplete list of values is given, those columns which are not specified must have a default value; if a value to be inserted in a primary key or unique index is the same as an existing one, the existing row is deleted and the new row inserted

RESET QUERY CACHE; remove all queries from the query cache

REVOKE <privilegelist> ON <db_name/tb_name/. . . > FROM 'user_name'@'host_name'; revoke specific privileges

SELECT <expression>; display the results of an expression

SELECT <col_name list> FROM tb_name WHERE <condition>; display rows and columns in a table that match the WHERE condition; the column name list can be replaced with * for all columns; omitting the WHERE condition results in all rows being displayed[4]

SELECT <col_name list> FROM tb_name WHERE <condition> GROUP BY col_name HAVING <condition>; display rows and columns in a table that match the WHERE condition, grouped by the entries in one column which satisfy the HAVING condition; the column name list can be replaced with * for all columns; omitting the WHERE condition results in all rows being displayed

SELECT <col_name list> FROM tb_name WHERE <condition> ORDER BY col_name; display rows and columns in a table that match the WHERE condition in the order of the entries in one column; the column name list can be replaced with * for all columns; omitting the WHERE condition results in all rows being displayed

SELECT <col_name list> INTO OUTFILE file_name WHERE <condition>; copy columns to an external CSV file; the column name list can be replaced with * for all columns; omitting the WHERE condition results in all rows being copied

SELECT col_name INTO DUMPFILE file_name WHERE <condition>; copy a single string of binary data to an external file; the WHERE condition cannot result in more than one string of binary data being selected or be omitted

[4]LIMIT may be used in all selections from a table to restrict the number of rows being displayed.

if its omission would lead to the same result

SET user_var_name=value; set the value of a user variable

SET col_name=value assign a value to a column

SET GLOBAL sys_var_name=value; set the value of a system variable = `SET @@global.sys_var_name=value;`

SET PASSWORD [FOR 'user_name'@'host_name'] = PASSWORD('password'); set or change a user's password, the current user if no user is specified

SET SESSION sys_var_name=value; set the value of a system variable for this session = `SET @@session.sys_var_name=value;`

SHOW CHARACTER SET; display the available character sets and their default collations[5]

SHOW COLLATION; display the available collations; collations in `.ci` are case insensitive, those in `.cs` are case sensitive and those in `.bin` are binary collations

SHOW [FULL] COLUMNS [FROM tb_name]; display columns; FULL displays the character set and user data

SHOW CREATE <object> <obj_name>; display the statement used to create the object which may be a DATABASE, EVENT, FUNCTION, PROCEDURE, TABLE, TRIGGER or VIEW

SHOW DATABASES; display database names, depending on the privilege level of the user

SHOW ENGINES; display available storage engines

SHOW ERRORS; display errors

SHOW EVENTS; display existing events

SHOW FUNCTION CODE; display function code

SHOW FUNCTION STATUS; display function status

SHOW GRANTS [FOR user]; display the privilege list [for a user]

[5]Most SHOW commands can take LIKE or WHERE conditions.

SHOW INDEX FROM tb_name; display the list of indexes on a table

SHOW OPEN TABLES; display open tables

SHOW PLUGINS; display available plugins

SHOW PRIVILEGES; display available privileges

SHOW PROCEDURE CODE; display procedure code

SHOW PROCEDURE STATUS; display procedure status

SHOW PROCESSLIST; show the status of running processes

SHOW PROFILE; show the way statements have been used during the current session

SHOW PROFILES; like SHOW PROFILE but gives more detailed information

SHOW STATUS; displays status variables

SHOW TABLES; displays tables in a database

SHOW TABLE STATUS; displays more detailed information about tables

SHOW TRIGGERS [FROM db_name]; display the existing triggers

SHOW VARIABLES; displays SESSION system variables; use SHOW GLOBAL VARIABLES to see global system variables

SHOW WARNINGS; displays current warnings; to show warnings automatically add

```
[client]
show_warnings
```

to `/etc/my.cnf`

SOURCE path/to/file_name; execute a batch file

TRUNCATE tb_name; =DELETE FROM `tb_name` but quicker because it creates an empty table and deletes the existing one rather than deleting rows; this can be useful for re-creating a corrupted table

UPDATE tb_name SET col_name1=value1[, colname2=value2,...] WHERE <condition>; updates individual entries in a row; omitting the WHERE condition changes every row in the column

USE db_name selects a database for use; does not require ';' terminator but must be the only command on a line

QUIT exit MariaDB; does not require '`;`' terminator

B Attributes

AUTO_INCREMENT increment the value in a column every time a row is added; only one individual column which must be indexed is permitted in a table; however, a secondary multiple column index may include an AUTO_INCREMENT column; an initial value may be specified; insert NULL into this column whenever a row is added

BINARY use binary collation on this column (in table creation/alteration)

BINARY col_name use case sensitivity on following column (=`col_name` COLLATE `Charset_Collation.bin`); see Appendix F

DEFAULT [col_name] inserts the default value where no value is specified[6]

DEFAULT def_value makes `def_value` the default value

NATIONAL specifies the default character set, i.e. `utf8`, for CHAR, VARCHAR data types

[6]See also DEFAULT() in Functions.

NCHAR = NATIONAL CHAR

NOT NULL column may not have NULL values

NULL column may have NULL values

PRIMARY use the primary key index when using HANDLER

PRIMARY KEY create a primary key on one or more columns; multiple instances of NULL are not permitted

UNIQUE create a unique index (multiple instances of NULL are permitted)

ZEROFILL create an UNSIGNED numeric column whose default value is 0

C Qualifiers

– unary minus

AFTER col_name insert a new or changed column after the specified column in a table

AFTER INSERT run trigger after INSERT statement

ALL return all the rows that match the expression — the default except in a UNION where the default is DISTINCT

ALL PRIVILEGES (in GRANT and REVOKE statements) = all the privileges at the specified level other than GRANT OPTION

AS create an alias for an expression using the form `<expression AS alias>`

ASC use ascending sort order on previous column (the default)

AVG_ROW_LENGTH value specifies the average row length for a table

b'binary_str' specify value of a BIT field in binary format

CHANGED only check Aria or MyISAM tables that have changed or have not been closed properly

CHARACTER SET charset_name use specified character set

CHECKSUM value when set to 1, maintain a checksum on each row of an Aria or MyISAM table

COLLATE collation_name use a specific collation; see Appendix F

COMMENT 'comment' add a comment of up to 64 characters to a table or stored routine

CONCURRENT load data into MyISAM tables concurrent with other actions

CROSS JOIN display every row in one table joined to every row in the other table

DATA DIRECTORY 'folder_name' specify the folder where Aria or MyISAM is to hold the table

DELAY_KEY_WRITE value if set to 1, delay key updates in Aria or MyISAM tables until the table is closed

DELAYED buffer data to be inserted or replaced when there is a break in which to write it (not available with INSERT/REPLACE ... SELECT commands, some storage engines or stored procedures); use sparingly and only where other clients may be accessing the table

DESC use descending sort order on previous column

DISABLE KEYS turn off key update (with ALTER TABLE)

DISTINCT ignore

- duplicate entries in a column, cf. LIMIT *n*
- duplicate rows in a SELECT response (the default in a UNION; use ALL if not desired)

DISTINCTROW = DISTINCT

ENABLE KEYS turn on key update (with ALTER TABLE)

ENGINE=engine_name specify the engine to be used with a table

EXTENDED do a full check or explanation; calculate checksums or repair indices row by row

FAST only check tables that have not been closed properly

FIELDS [OPTIONALLY] ENCLOSED BY 'char' without OPTIONALLY, specifies that all columns in an external data file are enclosed in single or double quotes or another character; with OPTIONALLY, specifies that CHAR, BINARY, TEXT or ENUM columns are enclosed in single or double quotes or another character

FIELDS ESCAPED BY 'char' specify the escape character (section E on page 44) in an external data file

FIELDS TERMINATED BY 'str'specifies the column terminator in an external data file

FIRST display the first row in an index when using HANDLER; insert a new or changed column first in a table

FORCE INDEX (index_name) use specified index for optimization, ORDER BY or GROUP BY

FROM source of data; where this is a subquery, it must be given an ALIAS

FULL display additional and metadata

GLOBAL display all variables; see SESSION

GROUP BY col_name groups rows in a table using the sorted order of the specified column; provides sub-totals in returns from COUNT() and the minima or maxima of groups in returns from MAX(), MIN(); alternative to ORDER BY for TEXT and BLOB columns; must precede ORDER BY if both are used

N.B. in a UNION the SELECT expressions must be in parentheses

HAVING condition used for WHERE conditions in GROUP BY expressions; can handle aggregate functions which are not possible with WHERE; must follow GROUP BY and precede ORDER BY

IDENTIFIED BY 'password' specifies password of user to whom privileges are being granted

IGNORE when altering a table, deleting, inserting, loading data or updating, replace out of range variables with maxima/minima, NULL with 0, 0000-00-00 or ' ' in a NOT NULL field and discard records that cause a duplicate key error; gives a warning when inserting but, when deleting, ignores errors but gives a warning; [actually turns off `sql_mode=STRICT_ALL_TABLES, NO_ZERO_IN_DATE, NO_ZERO_DATE`]; opposite of the REPLACE qualifier

IGNORE *n* **LINES** ignore the first *n* lines of an external data file being loaded into a table

IGNORE INDEX FOR <operation> (index_list) specify indexes to be ignored in a JOIN, ORDER BY or GROUP BY

INDEX DIRECTORY 'folder_name' specify the folder where Aria or MyISAM is to hold the index

INNER JOIN join in which only rows that have a match in both tables are displayed; note that a ON clause is required in standard SQL but not in MariaDB

INTO DUMPFILE file_name output file for a single item of binary, e.g. BLOB, data from a SELECT expression

INTO OUTFILE file_name output file for SELECT expression not involving binary data; file may be imported with LOAD DATA INFILE

LAST displays the last row in an index when using HANDLER

LEFT JOIN display the results from the table before and only the results from the table after that match the table before

LINES STARTING BY 'str' specifies prefix of rows in data files to be ignored

LINES TERMINATED BY 'str' specifies terminator of rows in data files — `'\r\n'`for DOS/Windows files

LIMIT *n* = LIMIT 0, *n* specifies the number of

- responses to a SELECT query to be displayed, cf. DISTINCT

  N.B. when used in a UNION, must be included in parentheses in the SELECT statement.
- instances of a delete instruction to be carried out
- instances of a HANDLER instruction to be carried out

LIMIT *m*,*n* *m* specifies the number of rows to be ignored and *n* the number of subsequent rows to be acted on

LIMIT *n* **OFFSET** *m* =LIMIT *m*,*n*

LINES STARTING BY ' ' specifies the characters which define the start of a row in an external data file

LINES TERMINATED BY ' ' specifies the characters which define the end of a row in an external data file

LOCAL may =

- current routine
- external data source is on the client host, not the server host
- SESSION

LOW_PRIORITY delays action on MyISAM table until no one is accessing the table (not recommended for normal use)

MAX_ROWS *n* specify the intended maximum number of rows to enable the storage engine optimise the table; does not limit number of rows used

MEDIUM do a standard check

NATURAL JOIN inner join or left join in which the columns in all tables are the same

NEXT displays the next row in an index when using HANDLER

ON expr may

- specify the databases, tables, columns,... on which a user has been granted privileges/has had privileges revoked
- specify the table on which to create an index
- specify the column(s) on which to create a JOIN
- specify the condition for a JOIN
- specify the condition for a JOIN followed by USING (column_list)
- specify an event within an UPDATE or stored procedure
- specify a table on which an event takes place within a stored procedure

ON DUPLICATE KEY UPDATE col_name=expr specifies the action to be taken when an insert generates a duplicate entry in an index; do not use with DELAYED or on tables with more than one unique index

ORDER BY col_name display or otherwise act on the rows in a table using the sorted order of this column; multiple columns are permitted; ASC is default; specify DESC if required; must follow GROUP BY and HAVING conditions

N.B. if used in a UNION, the SELECT expressions should be in parentheses and any aliases used in the SELECT statements must be used in the ORDER BY expression

ORDER BY expr display the output of a function in a particular order

ORDER BY RAND() display rows in a random order; use with LIMIT *n* to extract a random sample from a table

PREV displays the previous row in an index when using HANDLER

QUICK do not update indices when deleting; do not check links when checking; report checksums; repair indices only

REPLACE replace a record in a SELECT statement whose unique key is the same as the one being imported; opposite of the IGNORE qualifier

ROW_FORMAT specify whether table rows are stored in FIXED, DYNAMIC (variable length), COMPRESSED (MyISAM only) or PAGE (Aria only; an enhancement of DYNAMIC) format

SEPARATOR str specify the separator to be used in the output of GROUP_CONCAT()

SESSION display/set variable(s) applicable to this session only (the default)

SPATIAL create a spatial index (MyISAM and Aria only)

SQL_BIG_RESULT with GROUP BY or DISTINCT force the optimizer to use disk based, rather than memory based sorts; opposite of SQL_SMALL_RESULT

SQL_BUFFER_RESULT put the result in a temporary table

SQL_CACHE store the result in the query cache (even though query cache is off)

SQL_CALC_FOUND_ROWS calculates the number of rows that would have been returned in the absence of a LIMIT condition; must be in the first SELECT statement in a UNION; use UNION ALL in a UNION for an accurate response; use FOUND_ROWS() to display

SQL_NO_CACHE do not use a query cache on this SELECT (where query cache is on)

SQL_SMALL_RESULT with GROUP BY or DISTINCT force the optimizer to use memory based, rather than disk based sorts; opposite of SQL_BIG_RESULT

STRAIGHT_JOIN joins tables in the order specified in the SELECT expression

TEMPORARY create/drop a temporary table to be used while the connection is active leaving the actual table untouched

TO value use `value` for destination or new name

UNION combine the results of multiple SELECT expressions

UNSIGNED a numeric column which may only hold positive values

USE INDEX (index_list) specify indexes to be used in a JOIN

USING @var_name specify user variables in a prepared statement

USING charset_name specify the character set to be used for the output of CHAR() or CONVERT()

USING (col_name list) specify columns to be present in both tables in a JOIN

USING index_type specify index type

WHERE condition display only rows that match the condition

WITH QUERY EXPANSION perform a twin pass FULLTEXT search

WITH ROLLUP sums subtotals in a GROUP BY expression; cannot be used with ORDER BY

D Functions

D.1 Bit functions

& bitwise AND

<< converts a BIGINT to binary and shifts to the left

>> converts a BIGINT to binary and shifts to the right

BIT_COUNT(expr) returns the number of bits set in `expr`

^ bitwise XOR

| bitwise OR

~ bitwise NOT

D.2 Control flow functions

CASE value WHEN value1 THEN result1 [WHEN. . .]

ELSE result*n* END; returns `result1` when `value` matches `value1`,. . . ; `result`*n* if there is no match and NULL if there is no match and no ELSE

CASE WHEN condition1 THEN result1 [WHEN. . .]

ELSE result*n* END; returns the first `result` whose condition is satisfied, `result`*n* if no condition is satisfied and NULL if no condition is satisfied and there is no ELSE

IF(*expr1*,expr2,expr3) if `expr1` is TRUE, that is, NOT 0 and NOT NULL, do `expr2`, else do `expr3`; note that IF converts decimals to integers before evaluating

IFNULL(expr1,expr2) if `expr1` is NULL, returns `expr2`, else returns `expr1`

NULLIF(expr1,expr2) returns NULL if `expr1=expr2`, else returns `expr1`

D.3 Date functions

ADDDATE(date,*n*) returns the date with *n* days added; may also use DATE_ADD() syntax

ADDTIME(datetime,time) returns the `datetime+time`

CONVERT_TZ(datetime,zone1,zone2) returns the `zone2` datetime expression which matches the expression for `zone1` provided it is between 1970 and 2037

CURDATE() returns today's date as a date string or date value

CURRENT_DATE() = CURDATE()

CURTIME() returns the current time as a time string or time value; time values may take microsecond values

CURRENT_TIME() = CURTIME()

CURRENT_TIMESTAMP() = NOW()

DATE(datetime) returns the date part of a datetime expression

DATEDIFF(datetime1,datetime2) returns the number of days between the date in `datetime1` and the date in `datetime2`

DATE_ADD(datetime INTERVAL [+/-]period unit(s)) returns the date or datetime with the interval specified in `period unit(s)` added to/subtracted from `datetime`; ADDDATE() may take the same syntax

DATE FORMAT(datetime,format) returns a date or datetime expression in a particular format, for example:

%W	Weekday name	Sunday
%e	Day of month	1
%D	Day of month with suffix	1st
%M	Month name	January
%c	Month number	1
%Y	Year — four digit	2006
%T	24 hour clock	13:23:45

See https://mariadb.com/kb/en/date_format/ for full list.

DATE_SUB(datetime INTERVAL [+/-]period unit(s)) returns the date or datetime with the interval specified in `period unit(s)` subtracted from `datatime`; SUBDATE() may take the same syntax

DAYNAME() returns the day of a DATE as a weekday name

DAYOFMONTH() returns the day of a DATE as a number between 1 and 31

DAY() = DAYOFMONTH()

DAYOFWEEK() returns the day of a DATE as 1 for Sunday, 2 for Monday, etc.

DAYOFYEAR() returns the day of a DATE as a day number for the year

EXTRACT(unit(s) FROM datetime) returns the specified unit(s) of datetime expression

FROM DAYS(n) returns the date represented by n; only works with the Gregorian calendar

FROM UNIXTIME(t[,format]) returns the datetime represented by UNIX time t, optionally formatted using the same expressions as DATE_FORMAT

GET FORMAT(datetime,format) returns the datetime in USA, JIS, EUR, ISO or INTERNAL format

HOUR() returns the hour in a time expression

LAST DAY() returns the date of the last day of the month in a datetime expression

LOCALTIME() = NOW()

LOCALTIMESTAMP() = NOW()

MAKEDATE(year,n) returns the date of day number n in `year`

MAKETIME(h,m,s) returns the time using h, m and s as the values for the time

MICROSECOND() returns the microsecond part of a datetime expression

MINUTE() returns the minute in a time expression

MONTH() returns the month number in a date expression

MONTHNAME() returns the name of a month from a date expression

NOW() returns the current date and time, as at the start of the operation in which it is called; see SYSDATE()

PERIOD_ADD(expr,n) add n months to `expr` (expressed as YYMM or YYYYMM) and return result in YYYYMM format

PERIOD_DIFF(expr1,expr2) returns the number of months between the date in `expr1` and the date in `expr2` (both expressed as YYMM or YYYYMM)

QUARTER() returns the quarter of the year expressed as 1 to 4 from a date expression

SECOND() returns the seconds from a time expression

SEC_TO_TIME() converts a number of seconds into a time expression

STR_TO_DATE(str,format) converts `str` using a `format` as specified in DATE_FORMAT() to a datetime expression

SUB_DATE(date,*n*) returns the date expression with *n* days subtracted; may also use DATE_SUB() syntax

SUBTIME(datetime,time) returns the `datetime`−`time`

SYSDATE() returns the current date and time, at the time at which it is called; see NOW()

TIMEDIFF(datetime1,datetime2) returns the time difference between two datetime expressions

TIMESTAMP(datetime[,time]) returns a datetime expression as a timestamp, adding `time` to `datetime` if present

TIMESTAMPADD(unit,*n*,datetime) add *n* datetime units to `datetime`

TIMESTAMPDIFF(unit,datetime1,datetime2) returns the difference in units between `datetime1` and `datetime2`

TIME_FORMAT(time,format) returns a time only expression in a DATE_FORMAT() format relevant to time expressions

TIME_TO_SEC(time) returns the number of seconds in a time expression

TO_DAYS(date) returns the Gregorian day number of a date expression

TO_SECONDS(date) returns the number of seconds since 0000-00-00 in the Gregorian calendar

UNIX_TIMESTAMP[(datetime)] returns the number of seconds since 1970-01-01 00:00:00 or between 1970-01-01 00:00:00 and `datetime` if specified

UTC_DATE() returns the UTC date as a date expression

UTC_TIME() returns the UTC time as a time expression

UTC_TIMESTAMP() returns the UTC date and time as a timestamp expression

WEEK(datetime[,mode]) returns the week number of a datetime depending on the `mode` specified; see https://mariadb.com/kb/en/week/ for more information

WEEKDAY(date) returns 0 for Monday, 1 for Tuesday and so on for the day of a date expression

WEEKOFYEAR(datetime) = `WEEK(datetime,3)`

YEAR(datetime) returns the year part of a datetime as a number

YEARWEEK(datetime[,mode]) returns the year and week number of a date expression using the same modes as `WEEK(datetime[,mode])`

D.4 Dynamic columns functions

See https://mariadb.com/kb/en/dynamic-columns-functions/ for more information.

D.5 Encryption functions

AES_DECRYPT(str1,str2) decrypt `str1` using `str2` as the password

AES_ENCRYPT(str1,str2) encrypt `str1` using `str2` as the password

COMPRESS(str) compress a string (for use in a BLOB column)

DECODE(str1,str2) decode `str1` using `str2` as the password

DES_DECRYPT(str1[,key]) decrypt `str1`, using `key` if supplied

DES_ENCRYPT(str1[,key]) decrypt `str1`, using `key` if supplied; `key` may be a key number or a key string

ENCODE(str1,str2) encode `str1` using `str2` as the password

ENCRYPT(str[,salt]) encrypt using Unix encryption; not recommended

MD5(str) calculate a 128 bit MD5 checksum for `str`

OLD_PASSWORD(str) allows users of pre 4.1 versions of MariaDB to log on

PASSWORD('str') encrypts passwords using MariaDB encryption

SHA(str) =SHA1(`str`)

SHA1(str) calculate a 160 bit (RF3174) checksum for `str`

SHA2(str,hash_len) calculate using `hash_len` a checksum for `str`; `hash_len` may be 224, 256 (the default), 384 or 512

UNCOMPRESS(str) uncompress a string (from a BLOB column)

UNCOMPRESSED_LENGTH(str) return the uncompressed length of a compressed string

D.6 Functions and modifiers for use with GROUP BY

AVG([DISTINCT] expr) return the average value of the rows that match `expr`; use DISTINCT to exclude duplicate rows

BIT_AND(expr) return the bitwise AND of the `expr`

BIT_OR(expr) return the bitwise OR of the `expr`

BIT_XOR(expr) return the bitwise XOR of the `expr`

COUNT(expr) return the number of NOT NULL rows which match `expr`

COUNT(DISTINCT expr[,expr,. . .]) return the number of different rows which match the `expr`

GROUP_CONCAT([DISTINCT] expr[,expr,. . .]) returns a concatenated list of NOT NULL values which satisfy `expr`; use ORDER BY to manage the order of the values and SEPARATOR to separate values

MAX([DISTINCT] expr) display row holding highest value matching the `expr`

MIN([DISTINCT] expr) display row holding lowest value matching the `expr`

STD(expr) returns the population standard deviation of an `expr` =VAR_POP(`expr`)

STDDEV(expr) =STD(`expr`)

STDDEV_POP(expr) =STD(`expr`)

STDDEV_SAMP(expr) returns the sample standard deviation of an expression; = square root of VAR_SAMP(`expr`) [?]

SUM([DISTINCT] expr) returns the sum of the values matching `expr`

VARIANCE(expr) =VAR_POP()

VAR_POP(expr) returns the population standard variance using the number of rows as the population

VAR_SAMP(expr) returns the sample standard variance using the number of rows-1 as the denominator

D.7 Geographic functions

See https://mariadb.com/kb/en/geographic-functions/ for more information.

D.8 Information functions

BENCHMARK(n,expr) execute `expr` n times to allow comparison of query times

BINLOG_GTID_POS(binlog_filename,binlog_offset) return the GTID position corresponding to the older Binlog position

CHARSET(str) return the character set of `str`

COERCIBILITY(str) return the coercibility of `str`, showing whether it will be converted in the case of a collation conflict

COLLATION(str) return the collation of `str`

CONNECTION_ID() return the connection ID of the current connection

CURRENT USER() return the username of the user who has authenticated the current client; this may differ from USER() if for example a user has been granted access as an anonymous user

DATABASE() return the current database in use

FOUND_ROWS() return the number of rows found by SQL_CALC_FOUND_ROWS in a SELECT expression; useful if a LIMIT condition has restricted the number of rows displayed; however, if no rows are found, returns the previous value

LAST_INSERT_ID() return the most recent AUTO _INCREMENT value

LAST_INSERT_ID(expr) use `expr` for the next AUTO _INCREMENT value

ROW_COUNT() return the number of rows affected by the last statement

SCHEMA() = DATABASE()

SESSION_USER() = USER()

SYSTEM_USER() = USER()

USER() return the username of the current user, which may differ from CURRENT USER(), the user who has authenticated the current client

VERSION() returns the current version

[7] See also DEFAULT in Attributes

D.9 Miscellaneous functions

DEFAULT(col_name) return the default value of a column[7]

INET_ATON('dotted_quad') returns the value of `dotted_quad`; use unsigned integers for storage

INET_NTOA(n) returns the dotted quad representing n

LAST_VALUE(expr[,expr,...]) return the value of the last `expr` set

NAME_CONST('name',value) assign `name` to `value`

SLEEP(n) wait for n seconds

UUID() return the UUID

UUID_SHORT() return short version of the UUID

VALUES(col_name) sets the value of a column when updating using INSERT ... ON DUPLICATE KEY UPDATE

D.10 Numeric functions

+ add

/ divide

***** multiply

n**%**m modulo; returns the remainder of $\frac{n}{m}$

DIV integer division = FLOOR()

ABS(n) return absolute value of n

ACOS(n) return arc cosine of n

ASIN(n) return arc sine of n

ATAN(n) return arc tangent of n

ATAN2(y,x) return the arc tangent of $\frac{y}{x}$ using their signs to calculate the quadrant

CEIL(n) = CEILING()

CEILING(n) return the smallest integer not less than n

CONV(n,base1,base2) convert n from `base1` to `base2`; if `base2` is negative, n is treated as signed; available bases: 2 to 36

COS(r) return the cosine of r radians

COT(n) return the cotangent of n

CRC32(str) return the CRC value of `str`

DEGREES(r) return r radians as degrees

EXP(n) return e^n

FLOOR(n) return the greatest integer not more than n

LN(n) return the natural logarithm of n

LOG([b],n) return the logarithm of n to base b or the natural logarithm if b is omitted

LOG10(n) return the logarithm of n to base 10

LOG2(n) returns the logarithm of n to base 2; this also happens to be the number of bytes required to store the number

MOD(n,m) returns the remainder of $\frac{n}{m}$

n**MOD**m =MOD(n,m)

OCT(n) returns the octal value of n as a string

PI() returns the double precision value of π but normally displays only 6 decimal places

POW(x,y) return x^y

POWER(x,y) = POW(x,y)

RADIANS(d) return d degrees as radians

RAND([s]) return a random floating point number between 0 and 1, optionally using the integer s as the seed

ROUND(n[,d]) return n rounded to the nearest integer, or decimal place if d is specified; if d is negative round to 10^{-d}; use CEILING(), FLOOR() or TRUNCATE() for specific behaviours

SIGN(n) return the sign of n; -1 if negative, 0 if zero or 1 if positive

SIN(r) return the sine of r radians

SQRT(n) return the square root of a non-negative number n

TAN(r) return the tangent of r radians

TRUNCATE(n,d) return n truncated to d decimal places and rounded towards 0; if d is negative round to 10^{-d}

– subtract

D.11 String functions

ASCII(str) return a value between 0 and 255 representing first character of `str` where it is an 8-bit character; returns NULL if `str` is NULL or 0 if the string is an empty string

BIN(n) return a binary string representation of the BIGINT n

BINARY str cast `str` as a binary string; useful in case sensitive comparisons

BIT_LENGTH(expr) convert `expr` to a string and return its bit length

CAST(expr AS type) return `expr` in the specified `type` which may be

- BINARY[(N)]
- CHAR[(N)]
- DATE
- DATETIME[(D)]
- DECIMAL[(M[,D])]
- DOUBLE[(M[,D])]
- INTEGER =SIGNED [INTEGER]
- SIGNED [INTEGER]
- TIME[(D)]

- UNSIGNED [INTEGER]

CHAR(n,n,n,. . . [USING charset_name]) return a string of the characters represented by n,n,n. . . using `charset_name` or a binary string if `charset_name` is not specified

CHAR_LENGTH(str) returns the number of characters in a string regardless of whether they are single or multibyte characters; see LENGTH()

CHARACTER_LENGTH(str) = CHAR_LENGTH()

CONCAT(str1,str2,. . .) concatenate strings; returns NULL if any string is NULL; returns a binary string if any string is binary or numeric

CONCAT_WS(separator,str1,str2,. . .) concatenate strings using `separator` but skip NULL values; returns NULL if the separator is NULL

CONVERT(expr,type) =CAST(expr AS type)

CONVERT(expr USING charset_name) express `expr` using `charset_name`; not available for `ucs2`

ELT(n,str1,str2,str3,. . .) return the nth string; returns NULL if n is $<$ 1 or greater than the number of strings; inverse of FIELD()

EXPORT_SET((bits,on,off[,separator[,number_of_bits]]) return a string such that for every bit set in `bits`, you get an `on` string and for every bit not set in the value, you get an `off` string. Bits in `bits` are examined from right to left (from low-order to high-order bits). Strings are added to the result from left to right, separated by the separator string (the default being the comma character `","`). The number of bits examined is given by `number_of_bits` (defaults to 64). If any argument is NULL, returns NULL.

FIELD(str,str1,str2,str3,. . .) return the number of the first string which matches `str`; returns 0 if `str` is not found or `str` is NULL; inverse of ELT()

FIND_IN_SET(str,CSVstr) return the number of the first field containing `str` in a CSV row; returns 0 if `str` is not found or NULL if `str` or `CSVstr` is NULL; `str` should not contain a comma!

FORMAT(n,d) return a string containing n formatted with d decimal places

HEX(expr) return the hex string value of `expr` if `expr` is a number or the hex string value for each character in `expr` if `expr` is a string

INSERT(str1,p,l,str2) insert `str2` in `str1` at position p overwriting length l

INSTR(str1,str2) return the (first) position in `str1` where `str2` starts; opposite of LOCATE()

LCASE() = LOWER()

LEFT(str,n) return n characters from the left of `str`

LENGTH(str) return the number of bytes in a string; see CHAR_LENGTH()

expr LIKE 'pattern'
[ESCAPE 'escape_char'] compare `pattern` with a string, number, date or column; `pattern` may include the wildcards:

- `%` for any character (use `%` for 0 characters)
- `_` for a single character

To substitute the normal escape character `\` in `pattern`, add the optional ESCAPE character (section E on page 44).

To exclude particular `pattern` matches, use NOT LIKE.

LOAD_FILE(file_name) load a file as a string

LOCATE(str1,str2) return the position `str1` starts in `str2`; returns 0 if not found

LOCATE(str1,str2[,p]) return the position `str1` starts in `str2` , after position p if specified; returns 0 if not found; opposite of INSTR();

LOWER(str) return `str` in lower case `latin1`

LPAD(str1,l,str2) returns `str1` left padded with `str2` until its total length reaches l; if `str1` $> l$, returns `str1` reduced to l characters

LTRIM(str) return `str` with any spaces at the left of `str` removed

MAKE_SET((bits,str1,str2,...) return a set value (a string containing substrings separated by `","` characters) consisting of the strings that have the corresponding bit in `bits` set. `str1` corresponds to bit 0, `str2` to bit 1, and so on. NULL values in `str1`, `str2`, ... are not appended to the result.

MATCH(col_name list) AGAINST(expr[mode]) match `expr` against the columns indexed with a FULLTEXT index (MyISAM only); `mode` may be IN NATURAL LANGUAGE MODE (the default), IN BOOLEAN MODE (see table 4 on page 47) or WITH QUERY EXPANSION

MID(str,p,l) = SUBSTRING(str,p,l)

NOT LIKE see `expr` LIKE '`pattern`'

NOT REGEXP see `expr` REGEXP `pattern`

OCTET_LENGTH(str) = LENGTH(`str`)

ORD(str) returns the Unicode value of the first character in `str` (= ASCII(`str`) where the value is< 255)

POSITION(str1,str2) = LOCATE(`str1`,`str2`)

QUOTE(str) returns `str` in SQL compliant format

expr REGEXP pattern match `pattern` within `expr`

- **.** matches a single character
- **[abc]** matches `a`, `b` or `c`
- **[a-e]** matches `a`, `b`, `c`, `d` or `e`;
- **[0-9]** matches any numeral
- ***** matches any number of instances, e.g `x*` matches, `x`, `xx`, `xxx`, ...
- **^** matches the beginning of the expression
- **$** matches the end of the expression
- **{n}** matches n instances, e.g. '`9{6}`' matches `999999`

To exclude particular `pattern` matches, use NOT REGEXP.

expr RLIKE pattern = `expr` REGEXP `pattern`

REPEAT(str,n) returns `str` repeated n times

REPLACE(str,str1,str2) replaces `str1` in `str` with `str2`; always undertakes a case sensitive comparison

REVERSE(str) returns `str` in reverse order

RIGHT(str,l) returns l characters from the right of `str`

RPAD(str1,l,str2) right pads `str1` with `str2` until its length reaches l; if `str1` $> l$, returns `str1` reduced to l characters

RTRIM(str) return `str` with any spaces at the right of `str` removed

SOUNDEX(str) returns the English language soundex of `str`

expr1 SOUNDS LIKE expr2 = SOUNDEX(`expr1`)= SOUNDEX(`expr2`)

SPACE(n) return a string of n spaces

STRCMP(expr1,expr2) return the relative sort order of `expr1` to `expr2`; returns −1 if `expr1` is higher, 1 if `expr2` is higher and 0 if they are of equal

SUBSTR(str,p[,l]) = SUBSTRING(`str`,p[,l])

SUBSTRING(str,p[,l]) return the string, optionally of length l, in `str` starting from position p; if p is negative, p is calculated from the end of the string

SUBSTRING(str FROM p[FOR l]) =SUBSTRING(`str`,p[,l])

SUBSTRING_INDEX(str,d,n) returns the substring before (or after, if n is negative when it is counted from the end) the nth delimiter d

TRIM([{BOTH | LEADING | TRAILING}][str1 FROM] str2) remove spaces, or optionally `str1`, from BOTH ends of `str2`, if neither LEADING nor TRAILING are specified

UCASE(str) = UPPER(`str`)

UNHEX(h) returns the string value of the hex number h; opposite of HEX(`expr`)

UPPER(str) returns `str` in upper case `latin1`

D.12 Subquery functions

ALL(subquery) return 1 if `subquery` returns all rows

ANY(subquery) return 1 if `subquery` returns any rows

EXISTS(subquery) return 1 if `subquery` returns any rows

NOT EXISTS(subquery) return 1 if `subquery` returns no rows

SOME(subquery) =ANY(`subquery`)

E Escape characters

\0 ASCII 0, rather than the value zero

\' a single quote

\" a double quote

\N NULL

\Z ASCII 26 — to avoid confusion with Windows EOF character

\b backspace

\n new line, i.e. line feed

\r carriage return

\t tab character

\\ \

\% literal %, rather than wildcard %

\_ literal _ rather than wildcard _

F Operators

F.1 Arithmetic operators

+ add

DIV integer division = FLOOR()

/ divide

MOD modulo

% modulo

***** multiply

– subtract

F.2 Assignment operators

:= equal to; returns 1 if `true`; assign value on right to variable on left

= assign value on right to column on left

F.3 Comparison operators

<> != not equal to; returns 1 if true

< less than; returns 1 if true

<= lesser than or equal to; returns 1 if true

<=> NULL safe equal; returns 1 if both elements are NULL

= equal to; returns 1 if true;

> greater than; returns 1 if true

>= greater than or equal to; returns 1 if true

BETWEEN...AND returns 1 if `value` is equal to or greater than `min` and less than or equal to `max`

```
SELECT value BETWEEN min AND max;
```

Table 3: Operator precedence

BINARY, COLLATE
!
- (unary minus), ~(unary bit inversion)
^
*, /, DIV, %, MOD
+, -
<<, >>
&
|
=, <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
BETWEEN, CASE, WHEN, THEN, ELSE
NOT
&&, AND
||, OR, XOR
:=

COALESCE() returns the first non NULL value in a list or NULL if all values are NULL

GREATEST() returns highest value in a list or NULL if one value is NULL

expr IN (value1,value2,. . .) returns 1 if `expr` is in the value list or NULL if `expr` is not in the list and NULL is; IN performs automatic type conversion

INTERVAL(value,value1,value2,value3,. . .) returns −1 if `value` is NULL, 0 if `value` is less than `value1`, 1 if `value` is `value1` or less than `value2`, 2 if `value` is `value2` or less than `value3`, . . . ; `value1`, `value2`, `value3`, . . . integers must be in ascending order

IS boolean_value returns 1 if logically TRUE, i.e. the result of the comparison is >= 1; `boolean_value` may be TRUE, FALSE or UNKNOWN

IS NOT boolean_value returns 1 if logically true; `boolean_value` may be TRUE, FALSE or UNKNOWN

IS NOT NULL returns 1 if true, use instead of `<> NULL`

IS NULL returns 1 if true, use instead of `= NULL`

To find the row with the last AUTO_INCREMENT use

```
SELECT FROM tb_name WHERE col_name IS NULL;
```

To find '0000-00-00' in a NOT NULL DATE column use

```
SELECT FROM tb_name WHERE col_name IS NULL;
```

ISNULL() returns 1 if expression is NULL

LEAST() returns lowest value in a list or NULL if any value is NULL

NOT BETWEEN. . . AND returns 1 if `value` is less than `min` and greater than `max`

```
SELECT value NOT BETWEEN min AND max;
```

NOT expr IN (value1,value2,. . .) returns 1 if `expr` is not in the value list or NULL if `expr` is not in the list and NULL is; IN performs automatic type conversion

F.3.1 BOOLEAN MODE FULLTEXT operators

See table 4 on the following page for the BOOLEAN MODE FULLTEXT operators.

F.4 Logical operators

NOT ! logical NOT; returns 1 if evaluates to 0, NULL if NOT NULL

AND && logical AND; returns 1 if all operands are NOT 0 AND NOT NULL; returns 0 if one operand is 0; returns NULL if one operand is NULL and none is 0

XOR (a AND (NOT b)) OR ((NOT a) AND b) returns NULL if any operand is NULL; otherwise returns 1 if an odd number of operands is non zero and 0 if an even number of operands is non zero

OR || logical OR; returns 1 if one operand is NOT 0 AND NOT NULL; returns 0 if all operands are 0; returns NULL if one operand is NULL and the other is NULL OR 0

G Values

***** multiple columns wildcard

0 false

1 true

DUAL a dummy table name used where a FROM clause is required in a SELECT statement but no tables are referenced

Table 4: BOOLEAN MODE FULLTEXT operators

Operator	Description
`+`	The word is mandatory in all rows returned.
`-`	The word cannot appear in any row returned.
`<`	The word that follows has a lower relevance than other words, although rows containing it will still match
`>`	The word that follows has a higher relevance than other words.
`()`	Used to group words into subexpressions.
`~`	The word following contributes negatively to the relevance of the row (which is different to the '`-`' operator, which specifically excludes the word, or the '`<`' operator, which still causes the word to contribute positively to the relevance of the row.
`*`	The wildcard, indicating zero or more characters. It can only appear at the end of a word.
`"`	Anything enclosed in the double quotes is taken as a whole (so you can match phrases, for example).

FALSE	false
NULL	missing/unknown value; comes first in ascending sort order
TRUE	true