1.1.1.4.1 Selecting Data

The `SELECT` statement is used for retrieving data from tables, for select specific data, often based on a criteria given in the `WHERE` clause.

SELECT

SQL statement used primarily for retrieving data from a MariaDB database.

Joins & Subqueries

Documentation on the JOIN, UNION, EXCEPT and INTERSECT clauses, and on subqueries.

LIMIT

Documentation of the LIMIT clause.

ORDER BY

Order the results returned from a resultset.

GROUP BY

Aggregate data in a SELECT statement with the GROUP BY clause.

Common Table Expressions

Common table expressions are temporary named result sets.

SELECT WITH ROLLUP

Adds extra rows to the resultset that represent super-aggregate summaries

SELECT INTO OUTFILE

Write the resultset to a formatted file

SELECT INTO DUMPFILE

Write a binary string into file

FOR UPDATE

Acquires a lock on the rows

LOCK IN SHARE MODE

Acquires a write lock.

Optimizer Hints

Optimizer hints There are some options available in SELECT to affect the ex...

PROCEDURE

The PROCEDURE Clause of the SELECT Statement.

HANDLER

Direct access to reading rows from the storage engine.

DUAL

Dummy table name

SELECT ... OFFSET ... FETCH

Allows one to specify an offset, a number of rows to be returned, and wheth...

There are 2 related questions.

1.1.1.4.1.1 SELECT

Syntax

```
SELECT
    [ALL | DISTINCT | DISTINCTROW]
    [HIGH_PRIORITY]
    [STRAIGHT_JOIN]
    [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
    [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr ...]
    [ FROM table_references
      [WHERE where_condition]
      [GROUP BY {col_name | expr | position} [ASC | DESC], ... [WITH ROLLUP]]
      [HAVING where_condition]
      [ORDER BY {col_name | expr | position} [ASC | DESC], ...]
      [LIMIT {[offset,] row_count | row_count OFFSET offset  [ROWS EXAMINED rows_limit] } |
        [OFFSET start { ROW | ROWS }]
        [FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } { ONLY | WITH TIES }] ]
      procedure|[PROCEDURE procedure_name(argument_list)]
      [INTO OUTFILE 'file_name' [CHARACTER SET charset_name] [export_options] |
        INTO DUMPFILE 'file_name' | INTO var_name [, var_name] ]
      [FOR UPDATE lock_option | LOCK IN SHARE MODE lock_option]

export_options:
    [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
    ]

lock_option:
    [WAIT n | NOWAIT | SKIP LOCKED]
```

Contents

Description

`SELECT` is used to retrieve rows selected from one or more tables, and can include UNION statements and subqueries.

- Each *select_expr* expression indicates a column or data that you want to retrieve. You must have at least one select expression. See Select Expressions below.
- The `FROM` clause indicates the table or tables from which to retrieve rows. Use either a single table name or a `JOIN` expression. See JOIN for details. If no table is involved, FROM DUAL can be specified.
- Each table can also be specified as `db_name.tabl_name`. Each column can also be specified as `tbl_name.col_name` or even `db_name.tbl_name.col_name`. This allows one to write queries which involve multiple databases. See Identifier Qualifiers for syntax details.
- The `WHERE` clause, if given, indicates the condition or conditions that rows must satisfy to be selected.

`where_condition` is an expression that evaluates to true for each row to be selected. The statement selects all rows if there is no WHERE clause.
    - In the `WHERE` clause, you can use any of the functions and operators that MariaDB supports, except for aggregate (summary) functions. See Functions and Operators and Functions and Modifiers for use with GROUP BY (aggregate).
- Use the ORDER BY clause to order the results.
- Use the LIMIT clause allows you to restrict the results to only a certain number of rows, optionally with an offset.
- Use the GROUP BY and `HAVING` clauses to group rows together when they have columns or computed values in common.

SELECT can also be used to retrieve rows computed without reference to any table.

Select Expressions

A `SELECT` statement must contain one or more select expressions, separated by commas. Each select expression can be one of the following:

- The name of a column.
- Any expression using functions and operators.
- `*` to select all columns from all tables in the `FROM` clause.
- `tbl_name.*` to select all columns from just the table *tbl_name*.

When specifying a column, you can either use just the column name or qualify the column name with the name of the table using `tbl_name.col_name`. The qualified form is useful if you are joining multiple tables in the `FROM` clause. If you do not qualify the column names when selecting from multiple tables, MariaDB will try to find the column in each table. It is an error if that column name exists in multiple tables.

You can quote column names using backticks. If you are qualifying column names with table names, quote each part separately as `` `tbl_name`.`col_name` ``.

If you use any grouping functions in any of the select expressions, all rows in your results will be implicitly grouped, as if you had used `GROUP BY NULL`. `GROUP BY NULL` being an expression behaves specially such that the entire result set is treated as a group.

DISTINCT

A query may produce some identical rows. By default, all rows are retrieved, even when their values are the same. To explicitly specify that you want to retrieve identical rows, use the `ALL` option. If you want duplicates to be removed from the resultset, use the `DISTINCT` option. `DISTINCTROW` is a synonym for `DISTINCT`. See also COUNT DISTINCT and SELECT UNIQUE in Oracle mode.

INTO

The `INTO` clause is used to specify that the query results should be written to a file or variable.

- SELECT INTO OUTFILE - formatting and writing the result to an external file.
- SELECT INTO DUMPFILE - binary-safe writing of the unformatted results to an external file.
- SELECT INTO Variable - selecting and setting variables.

The reverse of `SELECT INTO OUTFILE` is LOAD DATA.

LIMIT

Restricts the number of returned rows. See LIMIT and LIMIT ROWS EXAMINED for details.

LOCK IN SHARE MODE/FOR UPDATE

See LOCK IN SHARE MODE and FOR UPDATE for details on the respective locking clauses.

OFFSET ... FETCH

MariaDB starting with 10.6

See SELECT ... OFFSET ... FETCH.

ORDER BY

Order a resultset. See ORDER BY for details.

PARTITION

Specifies to the optimizer which partitions are relevant for the query. Other partitions will not be read. See Partition Pruning and Selection for details.

PROCEDURE

Passes the whole result set to a C Procedure. See PROCEDURE and PROCEDURE ANALYSE (the only built-in procedure not requiring the server to be recompiled).

SKIP LOCKED

MariaDB starting with 10.6

The SKIP LOCKED clause was introduced in MariaDB 10.6.0.

This causes those rows that couldn't be locked (LOCK IN SHARE MODE or FOR UPDATE) to be excluded from the result set. An explicit `NOWAIT` is implied here. This is only implemented on InnoDB tables and ignored otherwise.

SQL_CALC_FOUND_ROWS

When `SQL_CALC_FOUND_ROWS` is used, then MariaDB will calculate how many rows would have been in the result, if there would be no LIMIT clause. The result can be found by calling the function FOUND_ROWS() in your next sql statement.

max_statement_time clause

By using max_statement_time in conjunction with SET STATEMENT, it is possible to limit the execution time of individual queries. For example:

```
SET STATEMENT max_statement_time=100 FOR
  SELECT field1 FROM table_name ORDER BY field1;
```

WAIT/NOWAIT

Set the lock wait timeout. See WAIT and NOWAIT.

Examples

```
SELECT f1,f2 FROM t1 WHERE (f3<=10) AND (f4='y');
```

See Getting Data from MariaDB (Beginner tutorial), or the various sub-articles, for more examples.

1.1.1.4.1.2 Joins & Subqueries

Documentation on the JOIN, UNION, EXCEPT and INTERSECT clauses, and on subqueries.

Joins

Querying from multiple tables.

Subqueries

Queries within queries.

UNION

Combine the results from multiple SELECT statements into a single result set.

EXCEPT
Subtraction of two result sets.

INTERSECT
Records that are present in both result sets will be included in the result of the operation.

Precedence Control in Table Operations
Controlling order of execution in SELECT, UNION, EXCEPT, and INTERSECT.

MINUS
Synonym for EXCEPT.

1.1.1.4.1.2.1 Joins

Articles about joins in MariaDB.

Joining Tables with JOIN Clauses
An introductory tutorial on using the JOIN clause.

More Advanced Joins
A more advanced tutorial on JOINs.

JOIN Syntax
Description MariaDB supports the following JOIN syntaxes for the table_refe...

Comma vs JOIN
A query to grab the list of phone numbers for clients who ordered in the la...

There are 1 related questions.

6.2.5 Joining Tables with JOIN Clauses

1.1.1.4.1.2.1.2 More Advanced Joins

Contents

This article is a follow up to the Introduction to JOINs page. If you're just getting started with JOINs, go through that page first and then come back here.

The Employee Database

Let us begin by using an example employee database of a fairly small family business, which does not anticipate expanding in the future.

First, we create the table that will hold all of the employees and their contact information:

```
CREATE TABLE `Employees` (
  `ID` TINYINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,
  `First_Name` VARCHAR(25) NOT NULL,
  `Last_Name` VARCHAR(25) NOT NULL,
  `Position` VARCHAR(25) NOT NULL,
  `Home_Address` VARCHAR(50) NOT NULL,
  `Home_Phone` VARCHAR(12) NOT NULL,
  PRIMARY KEY (`ID`)
) ENGINE=MyISAM;
```

Next, we add a few employees to the table:

```
INSERT INTO `Employees` (`First_Name`, `Last_Name`, `Position`, `Home_Address`, `Home_Phone`)
  VALUES
  ('Mustapha', 'Mond', 'Chief Executive Officer', '692 Promiscuous Plaza', '326-555-3492'),
  ('Henry', 'Foster', 'Store Manager', '314 Savage Circle', '326-555-3847'),
  ('Bernard', 'Marx', 'Cashier', '1240 Ambient Avenue', '326-555-8456'),
  ('Lenina', 'Crowne', 'Cashier', '281 Bumblepuppy Boulevard', '328-555-2349'),
  ('Fanny', 'Crowne', 'Restocker', '1023 Bokanovsky Lane', '326-555-6329'),
  ('Helmholtz', 'Watson', 'Janitor', '944 Soma Court', '329-555-2478');
```

Now, we create a second table, containing the hours which each employee clocked in and out during the week:

```
CREATE TABLE `Hours` (
  `ID` TINYINT(3) UNSIGNED NOT NULL,
  `Clock_In` DATETIME NOT NULL,
  `Clock_Out` DATETIME NOT NULL
) ENGINE=MyISAM;
```

Finally, although it is a lot of information, we add a full week of hours for each of the employees into the second table that we created:

```
INSERT INTO `Hours`
  VALUES
  ('1', '2005-08-08 07:00:42', '2005-08-08 17:01:36'),
  ('1', '2005-08-09 07:01:34', '2005-08-09 17:10:11'),
  ('1', '2005-08-10 06:59:56', '2005-08-10 17:09:29'),
  ('1', '2005-08-11 07:00:17', '2005-08-11 17:00:47'),
  ('1', '2005-08-12 07:02:29', '2005-08-12 16:59:12'),
  ('2', '2005-08-08 07:00:25', '2005-08-08 17:03:13'),
  ('2', '2005-08-09 07:00:57', '2005-08-09 17:05:09'),
  ('2', '2005-08-10 06:58:43', '2005-08-10 16:58:24'),
  ('2', '2005-08-11 07:01:58', '2005-08-11 17:00:45'),
  ('2', '2005-08-12 07:02:12', '2005-08-12 16:58:57'),
  ('3', '2005-08-08 07:00:12', '2005-08-08 17:01:32'),
  ('3', '2005-08-09 07:01:10', '2005-08-09 17:00:26'),
  ('3', '2005-08-10 06:59:53', '2005-08-10 17:02:53'),
  ('3', '2005-08-11 07:01:15', '2005-08-11 17:04:23'),
  ('3', '2005-08-12 07:00:51', '2005-08-12 16:57:52'),
  ('4', '2005-08-08 06:54:37', '2005-08-08 17:01:23'),
  ('4', '2005-08-09 06:58:23', '2005-08-09 17:00:54'),
  ('4', '2005-08-10 06:59:14', '2005-08-10 17:00:12'),
  ('4', '2005-08-11 07:00:49', '2005-08-11 17:00:34'),
  ('4', '2005-08-12 07:01:09', '2005-08-12 16:58:29'),
  ('5', '2005-08-08 07:00:04', '2005-08-08 17:01:43'),
  ('5', '2005-08-09 07:02:12', '2005-08-09 17:02:13'),
  ('5', '2005-08-10 06:59:39', '2005-08-10 17:03:37'),
  ('5', '2005-08-11 07:01:26', '2005-08-11 17:00:03'),
  ('5', '2005-08-12 07:02:15', '2005-08-12 16:59:02'),
  ('6', '2005-08-08 07:00:12', '2005-08-08 17:01:02'),
  ('6', '2005-08-09 07:03:44', '2005-08-09 17:00:00'),
  ('6', '2005-08-10 06:54:19', '2005-08-10 17:03:31'),
  ('6', '2005-08-11 07:00:05', '2005-08-11 17:02:57'),
  ('6', '2005-08-12 07:02:07', '2005-08-12 16:58:23');
```

Working with the Employee Database

Now that we have a cleanly structured database to work with, let us begin this tutorial by stepping up one notch from the last tutorial and filtering our information a little.

Filtering by Name

Earlier in the week, an anonymous employee reported that Helmholtz came into work almost four minutes late; to verify this, we will begin our investigation by filtering out employees whose first names are "Helmholtz":

```
SELECT
  `Employees`.`First_Name`,
  `Employees`.`Last_Name`,
  `Hours`.`Clock_In`,
  `Hours`.`Clock_Out`
FROM `Employees`
INNER JOIN `Hours` ON `Employees`.`ID` = `Hours`.`ID`
WHERE `Employees`.`First_Name` = 'Helmholtz';
```

The result:

```
+------------+-----------+---------------------+---------------------+
| First_Name | Last_Name | Clock_In            | Clock_Out           |
+------------+-----------+---------------------+---------------------+
| Helmholtz  | Watson    | 2005-08-08 07:00:12 | 2005-08-08 17:01:02 |
| Helmholtz  | Watson    | 2005-08-09 07:03:44 | 2005-08-09 17:00:00 |
| Helmholtz  | Watson    | 2005-08-10 06:54:19 | 2005-08-10 17:03:31 |
| Helmholtz  | Watson    | 2005-08-11 07:00:05 | 2005-08-11 17:02:57 |
| Helmholtz  | Watson    | 2005-08-12 07:02:07 | 2005-08-12 16:58:23 |
+------------+-----------+---------------------+---------------------+
5 rows in set (0.00 sec)
```

This is obviously more information than we care to trudge through, considering we only care about when he arrived past 7:00:59 on any given day within this week; thus, we need to add a couple more conditions to our WHERE clause.

Filtering by Name, Date and Time

In the following example, we will filter out all of the times which Helmholtz clocked in that were before 7:01:00 and during the work week that lasted from the 8th to the 12th of August:

```
SELECT
  `Employees`.`First_Name`,
  `Employees`.`Last_Name`,
  `Hours`.`Clock_In`,
  `Hours`.`Clock_Out`
FROM `Employees`
INNER JOIN `Hours` ON `Employees`.`ID` = `Hours`.`ID`
WHERE `Employees`.`First_Name` = 'Helmholtz'
AND DATE_FORMAT(`Hours`.`Clock_In`, '%Y-%m-%d') >= '2005-08-08'
AND DATE_FORMAT(`Hours`.`Clock_In`, '%Y-%m-%d') <= '2005-08-12'
AND DATE_FORMAT(`Hours`.`Clock_In`, '%H:%i:%S') > '07:00:59';
```

The result:

```
+------------+-----------+---------------------+---------------------+
| First_Name | Last_Name | Clock_In            | Clock_Out           |
+------------+-----------+---------------------+---------------------+
| Helmholtz  | Watson    | 2005-08-09 07:03:44 | 2005-08-09 17:00:00 |
| Helmholtz  | Watson    | 2005-08-12 07:02:07 | 2005-08-12 16:58:23 |
+------------+-----------+---------------------+---------------------+
2 rows in set (0.00 sec)
```

We have now, by merely adding a few more conditions, eliminated all of the irrelevant information; Helmholtz was late to work on the 9th and the 12th of August.

Displaying Total Work Hours per Day

Suppose you would like toâ€”based on the information stored in both of our tables in the employee databaseâ€”develop a quick list of the total hours each employee has worked for each day recorded; a simple way to estimate the time each employee worked per day is exemplified below:

```
SELECT
   `Employees`.`ID`,
   `Employees`.`First_Name`,
   `Employees`.`Last_Name`,
   `Hours`.`Clock_In`,
   `Hours`.`Clock_Out`,
DATE_FORMAT(`Hours`.`Clock_Out`, '%T')-DATE_FORMAT(`Hours`.`Clock_In`, '%T') AS 'Total_Hours'
FROM `Employees` INNER JOIN `Hours` ON `Employees`.`ID` = `Hours`.`ID`;
```

The result (limited by 10):

```
+----+------------+-----------+---------------------+---------------------+-------------+
| ID | First_Name | Last_Name | Clock_In            | Clock_Out           | Total_Hours |
+----+------------+-----------+---------------------+---------------------+-------------+
|  1 | Mustapha   | Mond      | 2005-08-08 07:00:42 | 2005-08-08 17:01:36 |          10 |
|  1 | Mustapha   | Mond      | 2005-08-09 07:01:34 | 2005-08-09 17:10:11 |          10 |
|  1 | Mustapha   | Mond      | 2005-08-10 06:59:56 | 2005-08-10 17:09:29 |          11 |
|  1 | Mustapha   | Mond      | 2005-08-11 07:00:17 | 2005-08-11 17:00:47 |          10 |
|  1 | Mustapha   | Mond      | 2005-08-12 07:02:29 | 2005-08-12 16:59:12 |           9 |
|  2 | Henry      | Foster    | 2005-08-08 07:00:25 | 2005-08-08 17:03:13 |          10 |
|  2 | Henry      | Foster    | 2005-08-09 07:00:57 | 2005-08-09 17:05:09 |          10 |
|  2 | Henry      | Foster    | 2005-08-10 06:58:43 | 2005-08-10 16:58:24 |          10 |
|  2 | Henry      | Foster    | 2005-08-11 07:01:58 | 2005-08-11 17:00:45 |          10 |
|  2 | Henry      | Foster    | 2005-08-12 07:02:12 | 2005-08-12 16:58:57 |           9 |
+----+------------+-----------+---------------------+---------------------+-------------+
10 rows in set (0.00 sec)
```

1.1.1.4.1.2.1.3 JOIN Syntax

Description

MariaDB supports the following `JOIN` syntaxes for the `table_references` part of `SELECT` statements and multiple-table `DELETE` and `UPDATE` statements:

```
table_references:
    table_reference [, table_reference] ...

table_reference:
    table_factor
  | join_table

table_factor:
    tbl_name [PARTITION (partition_list)]
        [query_system_time_period_specification] [[AS] alias] [index_hint_list]
  | table_subquery [query_system_time_period_specification] [AS] alias
  | ( table_references )
  | { ON table_reference LEFT OUTER JOIN table_reference
        ON conditional_expr }

join_table:
    table_reference [INNER | CROSS] JOIN table_factor [join_condition]
  | table_reference STRAIGHT_JOIN table_factor
  | table_reference STRAIGHT_JOIN table_factor ON conditional_expr
  | table_reference {LEFT|RIGHT} [OUTER] JOIN table_reference join_condition
  | table_reference NATURAL [{LEFT|RIGHT} [OUTER]] JOIN table_factor

join_condition:
    ON conditional_expr
  | USING (column_list)

query_system_time_period_specification:
    FOR SYSTEM_TIME AS OF point_in_time
  | FOR SYSTEM_TIME BETWEEN point_in_time AND point_in_time
  | FOR SYSTEM_TIME FROM point_in_time TO point_in_time
  | FOR SYSTEM_TIME ALL

point_in_time:
    [TIMESTAMP] expression
  | TRANSACTION expression

index_hint_list:
    index_hint [, index_hint] ...

index_hint:
    USE {INDEX|KEY}
      [{FOR {JOIN|ORDER BY|GROUP BY}] ([index_list])
  | IGNORE {INDEX|KEY}
      [{FOR {JOIN|ORDER BY|GROUP BY}] (index_list)
  | FORCE {INDEX|KEY}
      [{FOR {JOIN|ORDER BY|GROUP BY}] (index_list)

index_list:
    index_name [, index_name] ...
```

A table reference is also known as a join expression.

Each table can also be specified as `db_name.tabl_name`. This allows to write queries which involve multiple databases. See Identifier Qualifiers for syntax details.

The syntax of `table_factor` is extended in comparison with the SQL Standard. The latter accepts only `table_reference`, not a list of them inside a pair of parentheses.

This is a conservative extension if we consider each comma in a list of table_reference items as equivalent to an inner join. For example:

```
SELECT * FROM t1 LEFT JOIN (t2, t3, t4)
                 ON (t2.a=t1.a AND t3.b=t1.b AND t4.c=t1.c)
```

is equivalent to:

```
SELECT * FROM t1 LEFT JOIN (t2 CROSS JOIN t3 CROSS JOIN t4)
                 ON (t2.a=t1.a AND t3.b=t1.b AND t4.c=t1.c)
```

In MariaDB, `CROSS JOIN` is a syntactic equivalent to `INNER JOIN` (they can replace each other). In standard SQL, they are not equivalent. `INNER JOIN` is used with an `ON` clause, `CROSS JOIN` is used otherwise.

In general, parentheses can be ignored in join expressions containing only inner join operations. MariaDB also supports nested joins (see http://dev.mysql.com/doc/refman/5.1/en/nested-join-optimization.html).

See System-versioned tables for more information about `FOR SYSTEM_TIME` syntax.

Index hints can be specified to affect how the MariaDB optimizer makes use of indexes. For more information, see How to force query plans.

Examples

```
SELECT left_tbl.*
  FROM left_tbl LEFT JOIN right_tbl ON left_tbl.id = right_tbl.id
  WHERE right_tbl.id IS NULL;
```

1.1.1.4.1.2.1.4 Comma vs JOIN

A query to grab the list of phone numbers for clients who ordered in the last two weeks might be written in a couple of ways. Here are two:

```
SELECT *
FROM
  clients,
  orders,
  phoneNumbers
WHERE
  clients.id = orders.clientId
  AND clients.id = phoneNumbers.clientId
  AND orderPlaced >= NOW() - INTERVAL 2 WEEK;
```

```
SELECT *
FROM
  clients
  INNER JOIN orders ON clients.id = orders.clientId
  INNER JOIN phoneNumbers ON clients.id = phoneNumbers.clientId
WHERE
  orderPlaced >= NOW() - INTERVAL 2 WEEK;
```

Does it make a difference? Not much as written. But you should use the second form. Why?

- **Readability.** Once the WHERE clause contains more than two conditions, it becomes tedious to pick out the difference between business logic (only dates in the last two weeks) and relational logic (which fields relate clients to orders). Using the JOIN syntax with an ON clause makes the WHERE list shorter, and makes it very easy to see how tables relate to each other.
- **Flexibility.** Let's say we need to see all clients even if they don't have a phone number in the system. With the second version, it's easy; just change `INNER JOIN phoneNumbers` to `LEFT JOIN phoneNumbers`. Try that with the first version, and MySQL version 5.0.12+ will issue a syntax error because of the change in precedence between the comma operator and the JOIN keyword. The solution is to rearrange the FROM clause or add parentheses to override the precedence, and that quickly becomes frustrating.
- **Portability.** The changes in 5.0.12 were made to align with SQL:2003. If your queries use standard syntax, you will have an easier time switching to a different database should the need ever arise.

1.1.1.4.1.2.2 Subqueries

A subquery is a query nested in another query.

Scalar Subqueries

Subquery returning a single value.

Row Subqueries

Subquery returning a row.

Subqueries and ALL

Return true if the comparison returns true for each row, or the subquery returns no rows.

Subqueries and ANY

Return true if the comparison returns true for at least one row returned by the subquery.

Subqueries and EXISTS

Returns true if the subquery returns any rows.

Subqueries in a FROM Clause

Subqueries are more commonly placed in a WHERE clause, but can also form part of the FROM clause.

Subquery Optimizations

Articles about subquery optimizations in MariaDB.

Subqueries and JOINs

Rewriting subqueries as JOINs, and using subqueries instead of JOINs.

Subquery Limitations

There are a number of limitations regarding subqueries.

There are 1 related questions.

1.1.1.4.1.2.2.1 Scalar Subqueries

A scalar subquery is a subquery that returns a single value. This is the simplest form of a subquery, and can be used in most places a literal or single column value is valid.

The data type, length and character set and collation are all taken from the result returned by the subquery. The result of a subquery can always be NULL, that is, no result returned. Even if the original value is defined as NOT NULL, this is disregarded.

A subquery cannot be used where only a literal is expected, for example LOAD DATA INFILE expects a literal string containing the file name, and LIMIT requires a literal integer.

Examples

```
CREATE TABLE sq1 (num TINYINT);

CREATE TABLE sq2 (num TINYINT);

INSERT INTO sq1 VALUES (1);

INSERT INTO sq2 VALUES (10* (SELECT num FROM sq1));

SELECT * FROM sq2;
+------+
| num  |
+------+
|   10 |
+------+
```

Inserting a second row means the subquery is no longer a scalar, and this particular query is not valid:

```
INSERT INTO sq1 VALUES (2);

INSERT INTO sq2 VALUES (10* (SELECT num FROM sq1));
ERROR 1242 (21000): Subquery returns more than 1 row
```

No rows in the subquery, so the scalar is NULL:

```
INSERT INTO sq2 VALUES (10* (SELECT num FROM sq3 WHERE num='3'));

SELECT * FROM sq2;
+------+
| num  |
+------+
|   10 |
| NULL |
+------+
```

A more traditional scalar subquery, as part of a WHERE clause:

```
SELECT * FROM sq1 WHERE num = (SELECT MAX(num)/10 FROM sq2);
+------+
| num  |
+------+
|    1 |
+------+
```

1.1.1.4.1.2.2.2 Row Subqueries

A row subquery is a subquery returning a single row, as opposed to a scalar subquery, which returns a single column from a row, or a literal.

Examples

```
CREATE TABLE staff (name VARCHAR(10), age TINYINT);

CREATE TABLE customer (name VARCHAR(10), age TINYINT);

INSERT INTO staff VALUES ('Bilhah',37), ('Valerius',61), ('Maia',25);

INSERT INTO customer VALUES ('Thanasis',48), ('Valerius',61), ('Brion',51);

SELECT * FROM staff WHERE (name,age) = (SELECT name,age FROM customer WHERE name='Valerius');
+----------+------+
| name     | age  |
+----------+------+
| Valerius |   61 |
+----------+------+
```

Finding all rows in one table also in another:

```
SELECT name,age FROM staff WHERE (name,age) IN (SELECT name,age FROM customer);
+----------+------+
| name     | age  |
+----------+------+
| Valerius |   61 |
+----------+------+
```

1.1.1.4.1.2.2.3 Subqueries and ALL

Contents

Subqueries using the ALL keyword will return `true` if the comparison returns `true` for each row returned by the subquery, or the subquery returns no rows.

Syntax

```
scalar_expression comparison_operator ALL <Table subquery>
```

- `scalar_expression` may be any expression that evaluates to a single value
- `comparison_operator` may be any one of: `=`, `>`, `<`, `>=`, `<=`, `<>` or `!=`

`ALL` returns:

- `NULL` if the comparison operator returns `NULL` for at least one row returned by the Table subquery or scalar_expression returns `NULL`.
- `FALSE` if the comparison operator returns `FALSE` for at least one row returned by the Table subquery.
- `TRUE` if the comparison operator returns `TRUE` for all rows returned by the Table subquery, or if Table subquery returns no rows.

`NOT IN` is an alias for `<> ALL`.

Examples

```
CREATE TABLE sq1 (num TINYINT);

CREATE TABLE sq2 (num2 TINYINT);

INSERT INTO sq1 VALUES(100);

INSERT INTO sq2 VALUES(40),(50),(60);

SELECT * FROM sq1 WHERE num > ALL (SELECT * FROM sq2);
+------+
| num  |
+------+
|  100 |
+------+
```

Since `100` > all of `40`, `50` and `60`, the evaluation is true and the row is returned

Adding a second row to sq1, where the evaluation for that record is false:

```
INSERT INTO sq1 VALUES(30);

SELECT * FROM sq1 WHERE num > ALL (SELECT * FROM sq2);
+------+
| num  |
+------+
|  100 |
+------+
```

Adding a new row to sq2, causing all evaluations to be false:

```
INSERT INTO sq2 VALUES(120);

SELECT * FROM sq1 WHERE num > ALL (SELECT * FROM sq2);
Empty set (0.00 sec)
```

When the subquery returns no results, the evaluation is still true:

```
SELECT * FROM sq1 WHERE num > ALL (SELECT * FROM sq2 WHERE num2 > 300);
+------+
| num  |
+------+
|  100 |
|   30 |
+------+
```

Evaluating against a NULL will cause the result to be unknown, or not true, and therefore return no rows:

```
INSERT INTO sq2 VALUES (NULL);

SELECT * FROM sq1 WHERE num > ALL (SELECT * FROM sq2);
```

1.1.1.4.1.2.2.4 Subqueries and ANY

Contents

Subqueries using the ANY keyword will return `true` if the comparison returns `true` for at least one row returned by the subquery.

Syntax

The required syntax for an `ANY` or `SOME` quantified comparison is:

```
scalar_expression comparison_operator ANY <Table subquery>
```

Or:

```
scalar_expression comparison_operator SOME <Table subquery>
```

- `scalar_expression` may be any expression that evaluates to a single value.
- `comparison_operator` may be any one of `=`, `>`, `<`, `>=`, `<=`, `<>` or `!=`.

`ANY` returns:

- `TRUE` if the comparison operator returns `TRUE` for at least one row returned by the Table subquery.
- `FALSE` if the comparison operator returns `FALSE` for all rows returned by the Table subquery, or Table subquery has zero rows.
- `NULL` if the comparison operator returns `NULL` for at least one row returned by the Table subquery and doesn't returns `TRUE` for any of them, or if scalar_expression returns `NULL`.

`SOME` is a synmonym for `ANY`, and `IN` is a synonym for `= ANY`

Examples

```
CREATE TABLE sq1 (num TINYINT);

CREATE TABLE sq2 (num2 TINYINT);

INSERT INTO sq1 VALUES(100);

INSERT INTO sq2 VALUES(40),(50),(120);

SELECT * FROM sq1 WHERE num > ANY (SELECT * FROM sq2);
+------+
| num  |
+------+
|  100 |
+------+
```

`100` is greater than two of the three values, and so the expression evaluates as true.

SOME is a synonym for ANY:

```
SELECT * FROM sq1 WHERE num < SOME (SELECT * FROM sq2);
+------+
| num  |
+------+
|  100 |
+------+
```

`IN` is a synonym for `= ANY`, and here there are no matches, so no results are returned:

```
SELECT * FROM sq1 WHERE num IN (SELECT * FROM sq2);
Empty set (0.00 sec)
```

```
INSERT INTO sq2 VALUES(100);
Query OK, 1 row affected (0.05 sec)

SELECT * FROM sq1 WHERE num <> ANY (SELECT * FROM sq2);
+------+
| num  |
+------+
|  100 |
+------+
```

Reading this query, the results may be counter-intuitive. It may seem to read as "SELECT * FROM sq1 WHERE num does not match any results in sq2. Since it does match 100, it could seem that the results are incorrect. However, the query returns a result if the match does not match any *of* sq2. Since `100` already does not match `40`, the expression evaluates to true immediately, regardless of the 100's matching. It may be more easily readable to use SOME in a case such as this:

```
SELECT * FROM sq1 WHERE num <> SOME (SELECT * FROM sq2);
+------+
| num  |
+------+
|  100 |
+------+
```

1.1.1.4.1.2.2.5 Subqueries and EXISTS

Syntax

```
SELECT ... WHERE EXISTS <Table subquery>
```

Description

Subqueries using the `EXISTS` keyword will return `true` if the subquery returns any rows. Conversely, subqueries using `NOT EXISTS` will return `true` only if the subquery returns no rows from the table.

EXISTS subqueries ignore the columns specified by the SELECT of the subquery, since they're not relevant. For example,

```
SELECT col1 FROM t1 WHERE EXISTS (SELECT * FROM t2);
```

and

```
SELECT col1 FROM t1 WHERE EXISTS (SELECT col2 FROM t2);
```

produce identical results.

Examples

```
CREATE TABLE sq1 (num TINYINT);

CREATE TABLE sq2 (num2 TINYINT);

INSERT INTO sq1 VALUES(100);

INSERT INTO sq2 VALUES(40),(50),(60);

SELECT * FROM sq1 WHERE EXISTS (SELECT * FROM sq2 WHERE num2>50);
+------+
| num  |
+------+
|  100 |
+------+

SELECT * FROM sq1 WHERE NOT EXISTS (SELECT * FROM sq2 GROUP BY num2 HAVING MIN(num2)=40);
Empty set (0.00 sec)
```

1.1.1.4.1.2.2.6 Subqueries in a FROM Clause

Although subqueries are more commonly placed in a WHERE clause, they can also form part of the FROM clause. Such subqueries are commonly called derived tables.

If a subquery is used in this way, you must also use an AS clause to name the result of the subquery.

ORACLE mode

MariaDB starting with 10.6.0
From MariaDB 10.6.0, anonymous subqueries in a FROM clause (no AS clause) are permitted in ORACLE mode.

Examples

```
CREATE TABLE student (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student VALUES
  ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
  ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
  ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
  ('Tatiana', 'SQL', 87), ('Tatiana', 'Tuning', 83);
```

Assume that, given the data above, you want to return the average total for all students. In other words, the average of Chun's 148 (75+73), Esben's 74 (43+31), etc.

You cannot do the following:

```
SELECT AVG(SUM(score)) FROM student GROUP BY name;
ERROR 1111 (HY000): Invalid use of group function
```

A subquery in the FROM clause is however permitted:

```
SELECT AVG(sq_sum) FROM (SELECT SUM(score) AS sq_sum FROM student GROUP BY name) AS t;
+-------------+
| AVG(sq_sum) |
+-------------+
|    134.0000 |
+-------------+
```

From MariaDB 10.6 in ORACLE mode, the following is permitted:

```
SELECT * FROM (SELECT 1 FROM DUAL), (SELECT 2 FROM DUAL);
```

3.3.4.2 Subquery Optimizations

3.3.4.2.1 Subquery Optimizations Map

3.3.4.2.2 Semi-join Subquery Optimizations

3.3.4.2.3 Table Pullout Optimization

3.3.4.2.4 Non-semi-join Subquery Optimizations

3.3.4.2.5 Subquery Cache

3.3.4.2.6 Condition Pushdown Into IN subqueries

3.3.4.2.7 Conversion of Big IN Predicates Into Subqueries

3.3.4.2.8 EXISTS-to-IN Optimization

3.3.4.2.9 Optimizing GROUP BY and DISTINCE Clauses in Subqueries

1.1.1.4.1.2.2.8 Subqueries and JOINs

A subquery can quite often, but not in all cases, be rewritten as a JOIN.

Contents

Rewriting Subqueries as JOINS

A subquery using `IN` can be rewritten with the `DISTINCT` keyword, for example:

```
SELECT * FROM table1 WHERE col1 IN (SELECT col1 FROM table2);
```

can be rewritten as:

```
SELECT DISTINCT table1.* FROM table1, table2 WHERE table1.col1=table2.col1;
```

`NOT IN` or `NOT EXISTS` queries can also be rewritten. For example, these two queries returns the same result:

```
SELECT * FROM table1 WHERE col1 NOT IN (SELECT col1 FROM table2);
SELECT * FROM table1 WHERE NOT EXISTS (SELECT col1 FROM table2 WHERE table1.col1=table2.col1);
```

and both can be rewritten as:

```
SELECT table1.* FROM table1 LEFT JOIN table2 ON table1.id=table2.id WHERE table2.id IS NULL;
```

Subqueries that can be rewritten as a LEFT JOIN are sometimes more efficient.

Using Subqueries instead of JOINS

There are some scenarios, though, which call for subqueries rather than joins:

- When you want duplicates, but not false duplicates. Suppose `Table_1` has three rows — { `1` , `1` , `2` } — and `Table_2` has two rows — { `1` , `2` , `2` }. If you need to list the rows in `Table_1` which are also in `Table_2` , only this subquery-based `SELECT` statement will give the right answer (`1` , `1` , `2`):

```
SELECT Table_1.column_1
FROM   Table_1
WHERE  Table_1.column_1 IN
  (SELECT Table_2.column_1
    FROM   Table_2);
```

  This SQL statement won't work:

```
SELECT Table_1.column_1
FROM   Table_1,Table_2
WHERE  Table_1.column_1 = Table_2.column_1;
```

  because the result will be { `1` , `1` , `2` , `2` } — and the duplication of 2 is an error. This SQL statement won't work either:

```
SELECT DISTINCT Table_1.column_1
FROM   Table_1,Table_2
WHERE  Table_1.column_1 = Table_2.column_1;
```

  because the result will be { `1` , `2` } — and the removal of the duplicated 1 is an error too.

- When the outermost statement is not a query. The SQL statement:

```
UPDATE Table_1 SET column_1 = (SELECT column_1 FROM Table_2);
```

  can't be expressed using a join unless some rare SQL3 features are used.

- When the join is over an expression. The SQL statement:

```
SELECT * FROM Table_1
WHERE column_1 + 5 =
  (SELECT MAX(column_1) FROM Table_2);
```

  is hard to express with a join. In fact, the only way we can think of is this SQL statement:

```
SELECT Table_1.*
FROM   Table_1,
      (SELECT MAX(column_1) AS max_column_1 FROM Table_2) AS Table_2
WHERE  Table_1.column_1 + 5 = Table_2.max_column_1;
```

  which still involves a parenthesized query, so nothing is gained from the transformation.

- When you want to see the exception. For example, suppose the question is: what books are longer than Das Kapital? These two queries are effectively almost the same:

```
SELECT DISTINCT Bookcolumn_1.*
FROM   Books AS Bookcolumn_1 JOIN Books AS Bookcolumn_2 USING(page_count)
WHERE  title = 'Das Kapital';

SELECT DISTINCT Bookcolumn_1.*
FROM   Books AS Bookcolumn_1
WHERE  Bookcolumn_1.page_count >
  (SELECT DISTINCT page_count
  FROM   Books AS Bookcolumn_2
  WHERE  title = 'Das Kapital');
```

  The difference is between these two SQL statements is, if there are two editions of *Das Kapital* (with different page counts), then the self-join example will return the books which are longer than the shortest edition of *Das Kapital*. That might be the wrong answer, since the original question didn't ask for "... longer than `ANY` book named *Das Kapital*" (it seems to contain a false assumption that there's only one edition).

1.1.1.4.1.2.2.9 Subquery Limitations

Contents

There are a number of limitations regarding subqueries, which are discussed below.

The following tables and data will be used in the examples that follow:

```
CREATE TABLE staff(name VARCHAR(10),age TINYINT);

CREATE TABLE customer(name VARCHAR(10),age TINYINT);
```

```
INSERT INTO staff VALUES
('Bilhah',37), ('Valerius',61), ('Maia',25);

INSERT INTO customer VALUES
('Thanasis',48), ('Valerius',61), ('Brion',51);
```

ORDER BY and LIMIT

To use ORDER BY or limit LIMIT in subqueries both must be used.. For example:

```
SELECT * FROM staff WHERE name IN (SELECT name FROM customer ORDER BY name);
+----------+------+
| name     | age  |
+----------+------+
| Valerius |   61 |
+----------+------+
```

is valid, but

```
SELECT * FROM staff WHERE name IN (SELECT NAME FROM customer ORDER BY name LIMIT 1);
ERROR 1235 (42000): This version of MariaDB doesn't
  yet support 'LIMIT & IN/ALL/ANY/SOME subquery'
```

is not.

Modifying and Selecting from the Same Table

It's not possible to both modify and select from the same table in a subquery. For example:

```
DELETE FROM staff WHERE name = (SELECT name FROM staff WHERE age=61);
ERROR 1093 (HY000): Table 'staff' is specified twice, both
  as a target for 'DELETE' and as a separate source for data
```

Row Comparison Operations

There is only partial support for row comparison operations. The expression in

```
expr op {ALL|ANY|SOME} subquery,
```

must be scalar and the subquery can only return a single column.

However, because of the way `IN` is implemented (it is rewritten as a sequence of `=` comparisons and `AND`), the expression in

```
expression [NOT] IN subquery
```

is permitted to be an n-tuple and the subquery can return rows of n-tuples.

For example:

```
SELECT * FROM staff WHERE (name,age) NOT IN (
  SELECT name,age FROM customer WHERE age >=51]
);
+--------+------+
| name   | age  |
+--------+------+
| Bilhah |   37 |
| Maia   |   25 |
+--------+------+
```

is permitted, but

```
SELECT * FROM staff WHERE (name,age) = ALL (
  SELECT name,age FROM customer WHERE age >=51
);
ERROR 1241 (21000): Operand should contain 1 column(s)
```

is not.

Correlated Subqueries

Subqueries in the FROM clause cannot be correlated subqueries. They cannot be evaluated for each row of the outer query

since they are evaluated to produce a result set during when the query is executed.

Stored Functions

A subquery can refer to a stored function which modifies data. This is an extension to the SQL standard, but can result in indeterminate outcomes. For example, take:

```
SELECT ... WHERE x IN (SELECT f() ...);
```

where *f()* inserts rows. The function *f()* could be executed a different number of times depending on how the optimizer chooses to handle the query.

This sort of construct is therefore not safe to use in replication that is not row-based, as there could be different results on the master and the slave.

1.1.1.4.1.2.3 UNION

`UNION` is used to combine the results from multiple SELECT statements into a single result set.

Syntax

```
SELECT ...
UNION [ALL | DISTINCT] SELECT ...
[UNION [ALL | DISTINCT] SELECT ...]
[ORDER BY [column [, column ...]]]
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

Contents

Description

`UNION` is used to combine the results from multiple SELECT statements into a single result set.

The column names from the first `SELECT` statement are used as the column names for the results returned. Selected columns listed in corresponding positions of each SELECT statement should have the same data type. (For example, the first column selected by the first statement should have the same type as the first column selected by the other statements.)

If they don't, the type and length of the columns in the result take into account the values returned by all of the SELECTs, so there is no need for explicit casting. Note that currently this is not the case for recursive CTEs - see MDEV-12325.

Table names can be specified as `db_name.tbl_name`. This permits writing `UNION`s which involve multiple databases. See Identifier Qualifiers for syntax details.

UNION queries cannot be used with aggregate functions.

`EXCEPT` and `UNION` have the same operation precedence and `INTERSECT` has a higher precedence, unless running in Oracle mode, in which case all three have the same precedence.

ALL/DISTINCT

The `ALL` keyword causes duplicate rows to be preserved. The `DISTINCT` keyword (the default if the keyword is omitted) causes duplicate rows to be removed by the results.

UNION ALL and UNION DISTINCT can both be present in a query. In this case, UNION DISTINCT will override any UNION ALLs to its left.

MariaDB starting with 10.1.1

Until MariaDB 10.1.1, all `UNION ALL` statements required the server to create a temporary table. Since MariaDB

10.1.1, the server can in most cases execute `UNION ALL` without creating a temporary table, improving performance (see MDEV-334).

ORDER BY and LIMIT

Individual SELECTs can contain their own ORDER BY and LIMIT clauses. In this case, the individual queries need to be wrapped between parentheses. However, this does not affect the order of the UNION, so they only are useful to limit the record read by one SELECT.

The UNION can have global ORDER BY and LIMIT clauses, which affect the whole resultset. If the columns retrieved by individual SELECT statements have an alias (AS), the ORDER BY must use that alias, not the real column names.

HIGH_PRIORITY

Specifying a query as HIGH_PRIORITY will not work inside a UNION. If applied to the first SELECT, it will be ignored. Applying to a later SELECT results in a syntax error:

```
ERROR 1234 (42000): Incorrect usage/placement of 'HIGH_PRIORITY'
```

SELECT ... INTO ...

Individual SELECTs cannot be written INTO DUMPFILE or INTO OUTFILE. If the last SELECT statement specifies INTO DUMPFILE or INTO OUTFILE, the entire result of the UNION will be written. Placing the clause after any other SELECT will result in a syntax error.

If the result is a single row, SELECT ... INTO @var_name can also be used.

MariaDB starting with 10.4.0

Parentheses

From MariaDB 10.4.0, parentheses can be used to specify precedence. Before this, a syntax error would be returned.

Examples

`UNION` between tables having different column names:

```
(SELECT e_name AS name, email FROM employees)
UNION
(SELECT c_name AS name, email FROM customers);
```

Specifying the `UNION` 's global order and limiting total rows:

```
(SELECT name, email FROM employees)
UNION
(SELECT name, email FROM customers)
ORDER BY name LIMIT 10;
```

Adding a constant row:

```
(SELECT 'John Doe' AS name, 'john.doe@example.net' AS email)
UNION
(SELECT name, email FROM customers);
```

Differing types:

```
SELECT CAST('x' AS CHAR(1)) UNION SELECT REPEAT('y',4);
+----------------------+
| CAST('x' AS CHAR(1)) |
+----------------------+
| x                    |
| yyyy                 |
+----------------------+
```

Returning the results in order of each individual SELECT by use of a sort column:

```
(SELECT 1 AS sort_column, e_name AS name, email FROM employees)
UNION
(SELECT 2, c_name AS name, email FROM customers) ORDER BY sort_column;
```

Difference between UNION, EXCEPT and INTERSECT. `INTERSECT ALL` and `EXCEPT ALL` are available from MariaDB 10.5.0.

```
CREATE TABLE seqs (i INT);
INSERT INTO seqs VALUES (1),(2),(2),(3),(3),(4),(5),(6);

SELECT i FROM seqs WHERE i <= 3 UNION SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    5 |
|    6 |
+------+

SELECT i FROM seqs WHERE i <= 3 UNION ALL SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
|    2 |
|    3 |
|    3 |
|    3 |
|    3 |
|    4 |
|    5 |
|    6 |
+------+

SELECT i FROM seqs WHERE i <= 3 EXCEPT SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
+------+

SELECT i FROM seqs WHERE i <= 3 EXCEPT ALL SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
|    2 |
+------+

SELECT i FROM seqs WHERE i <= 3 INTERSECT SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    3 |
+------+

SELECT i FROM seqs WHERE i <= 3 INTERSECT ALL SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    3 |
|    3 |
+------+
```

Parentheses for specifying precedence, from MariaDB 10.4.0

```
CREATE OR REPLACE TABLE t1 (a INT);
CREATE OR REPLACE TABLE t2 (b INT);
CREATE OR REPLACE TABLE t3 (c INT);

INSERT INTO t1 VALUES (1),(2),(3),(4);
INSERT INTO t2 VALUES (5),(6);
INSERT INTO t3 VALUES (1),(6);

((SELECT a FROM t1) UNION (SELECT b FROM t2)) INTERSECT (SELECT c FROM t3);
+------+
| a    |
+------+
|    1 |
|    6 |
+------+

(SELECT a FROM t1) UNION ((SELECT b FROM t2) INTERSECT (SELECT c FROM t3));
+------+
| a    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    6 |
+------+
```

1.1.1.4.1.2.4 EXCEPT

The result of `EXCEPT` is all records of the left `SELECT` result set except records which are in right `SELECT` result set, i.e. it is subtraction of two result sets. From MariaDB 10.6.1, `MINUS` is a synonym when SQL_MODE=ORACLE is set.

Syntax

```
SELECT ...
(INTERSECT [ALL | DISTINCT] | EXCEPT [ALL | DISTINCT] | UNION [ALL | DISTINCT])
  SELECT ...
[(INTERSECT [ALL | DISTINCT] | EXCEPT [ALL | DISTINCT] | UNION [ALL | DISTINCT])
  SELECT ...]
[ORDER BY [{col_name | expr | position} [ASC | DESC]
  [, {col_name | expr | position} [ASC | DESC] ...]]]
[LIMIT {[offset,] row_count | row_count OFFSET offset}
| OFFSET start { ROW | ROWS }
| FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } { ONLY | WITH TIES } ]
```

Contents

Please note:

- Brackets for explicit operation precedence are not supported; use a subquery in the `FROM` clause as a workaround).

Description

MariaDB has supported `EXCEPT` and INTERSECT in addition to UNION since MariaDB 10.3.

The queries before and after `EXCEPT` must be SELECT or VALUES statements.

All behavior for naming columns, `ORDER BY` and `LIMIT` is the same as for `UNION`. Note that the alternative SELECT ... OFFSET ... FETCH syntax is only supported. This allows us to use the `WITH TIES` clause.

`EXCEPT` implicitly supposes a `DISTINCT` operation.

The result of `EXCEPT` is all records of the left `SELECT` result except records which are in right `SELECT` result set, i.e. it is subtraction of two result sets.

`EXCEPT` and `UNION` have the same operation precedence and `INTERSECT` has a higher precedence, unless running in Oracle mode, in which case all three have the same precedence.

MariaDB starting with 10.4.0

Parentheses

From MariaDB 10.4.0, parentheses can be used to specify precedence. Before this, a syntax error would be returned.

MariaDB starting with 10.5.0

ALL/DISTINCT

`EXCEPT ALL` and `EXCEPT DISTINCT` were introduced in MariaDB 10.5.0. The `ALL` operator leaves duplicates intact, while the `DISTINCT` operator removes duplicates. `DISTINCT` is the default behavior if neither operator is supplied, and the only behavior prior to MariaDB 10.5.

Examples

Show customers which are not employees:

```
(SELECT e_name AS name, email FROM customers)
EXCEPT
(SELECT c_name AS name, email FROM employees);
```

Difference between UNION, EXCEPT and INTERSECT. `INTERSECT ALL` and `EXCEPT ALL` are available from MariaDB 10.5.0.

```
CREATE TABLE seqs (i INT);
INSERT INTO seqs VALUES (1),(2),(2),(3),(3),(4),(5),(6);

SELECT i FROM seqs WHERE i <= 3 UNION SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    5 |
|    6 |
+------+

SELECT i FROM seqs WHERE i <= 3 UNION ALL SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
|    2 |
|    3 |
|    3 |
|    3 |
|    3 |
|    4 |
|    5 |
|    6 |
+------+

SELECT i FROM seqs WHERE i <= 3 EXCEPT SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
+------+

SELECT i FROM seqs WHERE i <= 3 EXCEPT ALL SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
|    2 |
+------+

SELECT i FROM seqs WHERE i <= 3 INTERSECT SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    3 |
+------+

SELECT i FROM seqs WHERE i <= 3 INTERSECT ALL SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    3 |
|    3 |
+------+
```

Parentheses for specifying precedence, from MariaDB 10.4.0

```
CREATE OR REPLACE TABLE t1 (a INT);
CREATE OR REPLACE TABLE t2 (b INT);
CREATE OR REPLACE TABLE t3 (c INT);

INSERT INTO t1 VALUES (1),(2),(3),(4);
INSERT INTO t2 VALUES (5),(6);
INSERT INTO t3 VALUES (1),(6);

((SELECT a FROM t1) UNION (SELECT b FROM t2)) EXCEPT (SELECT c FROM t3);
+------+
| a    |
+------+
|    2 |
|    3 |
|    4 |
|    5 |
+------+

(SELECT a FROM t1) UNION ((SELECT b FROM t2) EXCEPT (SELECT c FROM t3));
+------+
| a    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    5 |
+------+
```

Here is an example that makes use of the SEQUENCE storage engine and the VALUES statement, to generate a numeric sequence and remove some arbitrary numbers from it:

```
(SELECT seq FROM seq_1_to_10) EXCEPT VALUES (2), (3), (4);
+-----+
| seq |
+-----+
|   1 |
|   5 |
|   6 |
|   7 |
|   8 |
|   9 |
|  10 |
+-----+
```

1.1.1.4.1.2.5 INTERSECT

MariaDB starting with 10.3.0

INTERSECT was introduced in MariaDB 10.3.0.

The result of an intersect is the intersection of right and left `SELECT` results, i.e. only records that are present in both result sets will be included in the result of the operation.

Syntax

```
SELECT ...
(INTERSECT [ALL | DISTINCT] | EXCEPT [ALL | DISTINCT] | UNION [ALL | DISTINCT]) SELECT ...
[(INTERSECT [ALL | DISTINCT] | EXCEPT [ALL | DISTINCT] | UNION [ALL | DISTINCT]) SELECT ...]
[ORDER BY [column [, column ...]]]
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

Contents

Description

MariaDB has supported `INTERSECT` (as well as EXCEPT) in addition to UNION since MariaDB 10.3.

All behavior for naming columns, `ORDER BY` and `LIMIT` is the same as for UNION.

`INTERSECT` implicitly supposes a `DISTINCT` operation.

The result of an intersect is the intersection of right and left `SELECT` results, i.e. only records that are present in both result sets will be included in the result of the operation.

`INTERSECT` has higher precedence than `UNION` and `EXCEPT` (unless running running in Oracle mode, in which case all three have the same precedence). If possible it will be executed linearly but if not it will be translated to a subquery in the `FROM` clause:

```
(select a,b from t1)
union
(select c,d from t2)
intersect
(select e,f from t3)
union
(select 4,4);
```

will be translated to:

```
(select a,b from t1)
union
select c,d from
  ((select c,d from t2)
   intersect
   (select e,f from t3)) dummy_subselect
union
(select 4,4)
```

MariaDB starting with 10.4.0

Parentheses

From MariaDB 10.4.0, parentheses can be used to specify precedence. Before this, a syntax error would be returned.

MariaDB starting with 10.5.0

ALL/DISTINCT

`INTERSECT ALL` and `INTERSECT DISTINCT` were introduced in MariaDB 10.5.0. The `ALL` operator leaves duplicates intact, while the `DISTINCT` operator removes duplicates. `DISTINCT` is the default behavior if neither operator is supplied, and the only behavior prior to MariaDB 10.5.

Examples

Show customers which are employees:

```
(SELECT e_name AS name, email FROM employees)
INTERSECT
(SELECT c_name AS name, email FROM customers);
```

Difference between UNION, EXCEPT and INTERSECT. `INTERSECT ALL` and `EXCEPT ALL` are available from MariaDB

10.5.0.

```
CREATE TABLE seqs (i INT);
INSERT INTO seqs VALUES (1),(2),(2),(3),(3),(4),(5),(6);

SELECT i FROM seqs WHERE i <= 3 UNION SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    5 |
|    6 |
+------+

SELECT i FROM seqs WHERE i <= 3 UNION ALL SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
|    2 |
|    3 |
|    3 |
|    3 |
|    3 |
|    4 |
|    5 |
|    6 |
+------+

SELECT i FROM seqs WHERE i <= 3 EXCEPT SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
+------+

SELECT i FROM seqs WHERE i <= 3 EXCEPT ALL SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
|    2 |
+------+

SELECT i FROM seqs WHERE i <= 3 INTERSECT SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    3 |
+------+

SELECT i FROM seqs WHERE i <= 3 INTERSECT ALL SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    3 |
|    3 |
+------+
```

Parentheses for specifying precedence, from MariaDB 10.4.0

```
CREATE OR REPLACE TABLE t1 (a INT);
CREATE OR REPLACE TABLE t2 (b INT);
CREATE OR REPLACE TABLE t3 (c INT);

INSERT INTO t1 VALUES (1),(2),(3),(4);
INSERT INTO t2 VALUES (5),(6);
INSERT INTO t3 VALUES (1),(6);

((SELECT a FROM t1) UNION (SELECT b FROM t2)) INTERSECT (SELECT c FROM t3);
+------+
| a    |
+------+
|    1 |
|    6 |
+------+

(SELECT a FROM t1) UNION ((SELECT b FROM t2) INTERSECT (SELECT c FROM t3));
+------+
| a    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    6 |
+------+
```

1.1.1.4.1.2.6 Precedence Control in Table Operations

MariaDB starting with 10.4.0

Beginning in MariaDB 10.4, you can control the ordering of execution on table operations using parentheses.

Syntax

```
(  expression )
[ORDER BY [column[, column...]]]
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

Contents

Description

Using parentheses in your SQL allows you to control the order of execution for `SELECT` statements and Table Value Constructor, including `UNION`, `EXCEPT`, and `INTERSECT` operations. MariaDB executes the parenthetical expression before the rest of the statement. You can then use `ORDER BY` and `LIMIT` clauses the further organize the result-set.

Note: In practice, the Optimizer may rearrange the exact order in which MariaDB executes different parts of the statement. When it calculates the result-set, however, it returns values as though the parenthetical expression were executed first.

Example

```
CREATE TABLE test.t1 (num INT);

INSERT INTO test.t1 VALUES (1),(2),(3);

(SELECT * FROM test.t1
 UNION
 VALUES (10))
INTERSECT
VALUES (1),(3),(10),(11);
+------+
| num  |
+------+
|    1 |
|    3 |
|   10 |
+------+

((SELECT * FROM test.t1
  UNION
  VALUES (10))
 INTERSECT
 VALUES (1),(3),(10),(11))
ORDER BY 1 DESC;
+------+
| num  |
+------+
|   10 |
|    3 |
|    1 |
+------+
```

1.1.1.4.1.2.7 MINUS

MariaDB starting with 10.6.1

`MINUS` was introduced as a synonym for EXCEPT from MariaDB 10.6.1 when SQL_MODE=ORACLE is set.

```
CREATE TABLE seqs (i INT);
INSERT INTO seqs VALUES (1),(2),(2),(3),(3),(4),(5),(6);

SET SQL_MODE='ORACLE';

SELECT i FROM seqs WHERE i <= 3 MINUS SELECT i FROM seqs WHERE i>=3;
+------+
| i    |
+------+
|    1 |
|    2 |
+------+
```

1.1.1.4.1.3 LIMIT

Contents

Description

Use the `LIMIT` clause to restrict the number of returned rows. When you use a single integer *n* with `LIMIT`, the first *n* rows will be returned. Use the ORDER BY clause to control which rows come first. You can also select a number of rows after an offset using either of the following:

```
LIMIT offset, row_count
LIMIT row_count OFFSET offset
```

When you provide an offset *m* with a limit *n*, the first *m* rows will be ignored, and the following *n* rows will be returned.

Executing an UPDATE with the `LIMIT` clause is not safe for replication. `LIMIT 0` is an exception to this rule (see MDEV-6170).

There is a LIMIT ROWS EXAMINED optimization which provides the means to terminate the execution of SELECT statements which examine too many rows, and thus use too many resources. See LIMIT ROWS EXAMINED.

Multi-Table Updates

Until MariaDB 10.3.1, it was not possible to use `LIMIT` (or ORDER BY) in a multi-table UPDATE statement. This restriction was lifted in MariaDB 10.3.2.

GROUP_CONCAT

Starting from MariaDB 10.3.3, it is possible to use `LIMIT` with GROUP_CONCAT().

Examples

```
CREATE TABLE members (name VARCHAR(20));
INSERT INTO members VALUES('Jagdish'),('Kenny'),('Rokurou'),('Immaculada');

SELECT * FROM members;
+------------+
| name       |
+------------+
| Jagdish    |
| Kenny      |
| Rokurou    |
| Immaculada |
+------------+
```

Select the first two names (no ordering specified):

```
SELECT * FROM members LIMIT 2;
+---------+
| name    |
+---------+
| Jagdish |
| Kenny   |
+---------+
```

All the names in alphabetical order:

```
SELECT * FROM members ORDER BY name;
+------------+
| name       |
+------------+
| Immaculada |
| Jagdish    |
| Kenny      |
| Rokurou    |
+------------+
```

The first two names, ordered alphabetically:

```
SELECT * FROM members ORDER BY name LIMIT 2;
+------------+
| name       |
+------------+
| Immaculada |
| Jagdish    |
+------------+
```

The third name, ordered alphabetically (the first name would be offset zero, so the third is offset two):

```
SELECT * FROM members ORDER BY name LIMIT 2,1;
+-------+
| name  |
+-------+
| Kenny |
+-------+
```

From MariaDB 10.3.2, `LIMIT` can be used in a multi-table update:

```
CREATE TABLE warehouse (product_id INT, qty INT);
INSERT INTO warehouse VALUES (1,100),(2,100),(3,100),(4,100);

CREATE TABLE store (product_id INT, qty INT);
INSERT INTO store VALUES (1,5),(2,5),(3,5),(4,5);

UPDATE warehouse,store SET warehouse.qty = warehouse.qty-2, store.qty = store.qty+2
  WHERE (warehouse.product_id = store.product_id AND store.product_id  >= 1)
    ORDER BY store.product_id DESC LIMIT 2;

SELECT * FROM warehouse;
+------------+------+
| product_id | qty  |
+------------+------+
|          1 |  100 |
|          2 |  100 |
|          3 |   98 |
|          4 |   98 |
+------------+------+

SELECT * FROM store;
+------------+------+
| product_id | qty  |
+------------+------+
|          1 |    5 |
|          2 |    5 |
|          3 |    7 |
|          4 |    7 |
+------------+------+
```

From MariaDB 10.3.3, `LIMIT` can be used with GROUP_CONCAT, so, for example, given the following table:

```
CREATE TABLE d (dd DATE, cc INT);

INSERT INTO d VALUES ('2017-01-01',1);
INSERT INTO d VALUES ('2017-01-02',2);
INSERT INTO d VALUES ('2017-01-04',3);
```

the following query:

```
SELECT SUBSTRING_INDEX(GROUP_CONCAT(CONCAT_WS(":",dd,cc) ORDER BY cc DESC),",",1) FROM d;
+-----------------------------------------------------------------------------+
| SUBSTRING_INDEX(GROUP_CONCAT(CONCAT_WS(":",dd,cc) ORDER BY cc DESC),",",1) |
+-----------------------------------------------------------------------------+
| 2017-01-04:3                                                                |
+-----------------------------------------------------------------------------+
```

can be more simply rewritten as:

```
SELECT GROUP_CONCAT(CONCAT_WS(":",dd,cc) ORDER BY cc DESC LIMIT 1) FROM d;
+-----------------------------------------------------------+
| GROUP_CONCAT(CONCAT_WS(":",dd,cc) ORDER BY cc DESC LIMIT 1) |
+-----------------------------------------------------------+
| 2017-01-04:3                                              |
+-----------------------------------------------------------+
```

1.1.1.4.1.4 ORDER BY

Contents

Description

Use the `ORDER BY` clause to order a resultset, such as that are returned from a SELECT statement. You can specify just a column or use any expression with functions. If you are using the `GROUP BY` clause, you can use grouping functions in `ORDER BY`. Ordering is done after grouping.

You can use multiple ordering expressions, separated by commas. Rows will be sorted by the first expression, then by the second expression if they have the same value for the first, and so on.

You can use the keywords `ASC` and `DESC` after each ordering expression to force that ordering to be ascending or descending, respectively. Ordering is ascending by default.

You can also use a single integer as the ordering expression. If you use an integer *n*, the results will be ordered by the *n*th column in the select expression.

When string values are compared, they are compared as if by the STRCMP function. `STRCMP` ignores trailing whitespace and may normalize characters and ignore case, depending on the collation in use.

Duplicated entries in the `ORDER BY` clause are removed.

`ORDER BY` can also be used to order the activities of a DELETE or UPDATE statement (usually with the LIMIT clause).

MariaDB starting with 10.3.2

Until MariaDB 10.3.1, it was not possible to use `ORDER BY` (or LIMIT) in a multi-table UPDATE statement. This restriction was lifted in MariaDB 10.3.2.

MariaDB starting with 10.5

From MariaDB 10.5, MariaDB allows packed sort keys and values of non-sorted fields in the sort buffer. This can make filesort temporary files much smaller when VARCHAR, CHAR or BLOBs are used, notably speeding up some ORDER BY sorts.

Examples

```
CREATE TABLE seq (i INT, x VARCHAR(1));
INSERT INTO seq VALUES (1,'a'), (2,'b'), (3,'b'), (4,'f'), (5,'e');

SELECT * FROM seq ORDER BY i;
+------+------+
| i    | x    |
+------+------+
|    1 | a    |
|    2 | b    |
|    3 | b    |
|    4 | f    |
|    5 | e    |
+------+------+

SELECT * FROM seq ORDER BY i DESC;
+------+------+
| i    | x    |
+------+------+
|    5 | e    |
|    4 | f    |
|    3 | b    |
|    2 | b    |
|    1 | a    |
+------+------+

SELECT * FROM seq ORDER BY x,i;
+------+------+
| i    | x    |
+------+------+
|    1 | a    |
|    2 | b    |
|    3 | b    |
|    5 | e    |
|    4 | f    |
+------+------+
```

ORDER BY in an UPDATE statement, in conjunction with LIMIT:

```
UPDATE seq SET x='z' WHERE x='b' ORDER BY i DESC LIMIT 1;

SELECT * FROM seq;
+------+------+
| i    | x    |
+------+------+
|    1 | a    |
|    2 | b    |
|    3 | z    |
|    4 | f    |
|    5 | e    |
+------+------+
```

From MariaDB 10.3.2, `ORDER BY` can be used in a multi-table update:

```
CREATE TABLE warehouse (product_id INT, qty INT);
INSERT INTO warehouse VALUES (1,100),(2,100),(3,100),(4,100);

CREATE TABLE store (product_id INT, qty INT);
INSERT INTO store VALUES (1,5),(2,5),(3,5),(4,5);

UPDATE warehouse,store SET warehouse.qty = warehouse.qty-2, store.qty = store.qty+2
  WHERE (warehouse.product_id = store.product_id AND store.product_id  >= 1)
    ORDER BY store.product_id DESC LIMIT 2;

SELECT * FROM warehouse;
+------------+------+
| product_id | qty  |
+------------+------+
|          1 |  100 |
|          2 |  100 |
|          3 |   98 |
|          4 |   98 |
+------------+------+

SELECT * FROM store;
+------------+------+
| product_id | qty  |
+------------+------+
|          1 |    5 |
|          2 |    5 |
|          3 |    7 |
|          4 |    7 |
+------------+------+
```

1.1.1.4.1.5 GROUP BY

Contents

Use the `GROUP BY` clause in a SELECT statement to group rows together that have the same value in one or more column, or the same computed value using expressions with any functions and operators except grouping functions. When you use a `GROUP BY` clause, you will get a single result row for each group of rows that have the same value for the expression given in `GROUP BY`.

When grouping rows, grouping values are compared as if by the `=` operator. For string values, the `=` operator ignores trailing whitespace and may normalize characters and ignore case, depending on the collation in use.

You can use any of the grouping functions in your select expression. Their values will be calculated based on all the rows that have been grouped together for each result row. If you select a non-grouped column or a value computed from a non-grouped column, it is undefined which row the returned value is taken from. This is not permitted if the `ONLY_FULL_GROUP_BY` SQL_MODE is used.

You can use multiple expressions in the `GROUP BY` clause, separated by commas. Rows are grouped together if they match on each of the expressions.

You can also use a single integer as the grouping expression. If you use an integer *n*, the results will be grouped by the *n*th column in the select expression.

The `WHERE` clause is applied before the `GROUP BY` clause. It filters non-aggregated rows before the rows are grouped together. To filter grouped rows based on aggregate values, use the `HAVING` clause. The `HAVING` clause takes any expression and evaluates it as a boolean, just like the `WHERE` clause. You can use grouping functions in the `HAVING` clause. As with the select expression, if you reference non-grouped columns in the `HAVING` clause, the behavior is undefined.

By default, if a `GROUP BY` clause is present, the rows in the output will be sorted by the expressions used in the `GROUP BY`. You can also specify `ASC` or `DESC` (ascending, descending) after those expressions, like in ORDER BY. The default is `ASC`.

If you want the rows to be sorted by another field, you can add an explicit ORDER BY. If you don't want the result to be ordered, you can add ORDER BY NULL.

WITH ROLLUP

The `WITH ROLLUP` modifer adds extra rows to the resultset that represent super-aggregate summaries. For a full description with examples, see SELECT WITH ROLLUP.

GROUP BY Examples

Consider the following table that records how many times each user has played and won a game:

```
CREATE TABLE plays (name VARCHAR(16), plays INT, wins INT);
INSERT INTO plays VALUES
  ("John", 20, 5),
  ("Robert", 22, 8),
  ("Wanda", 32, 8),
  ("Susan", 17, 3);
```

Get a list of win counts along with a count:

```
SELECT wins, COUNT(*) FROM plays GROUP BY wins;
+------+----------+
| wins | COUNT(*) |
+------+----------+
|    3 |        1 |
|    5 |        1 |
|    8 |        2 |
+------+----------+
3 rows in set (0.00 sec)
```

The `GROUP BY` expression can be a computed value, and can refer back to an identifer specified with `AS`. Get a list of win averages along with a count:

```
SELECT (wins / plays) AS winavg, COUNT(*) FROM plays GROUP BY winavg;
+--------+----------+
| winavg | COUNT(*) |
+--------+----------+
| 0.1765 |        1 |
| 0.2500 |        2 |
| 0.3636 |        1 |
+--------+----------+
3 rows in set (0.00 sec)
```

You can use any grouping function in the select expression. For each win average as above, get a list of the average play count taken to get that average:

```
SELECT (wins / plays) AS winavg, AVG(plays) FROM plays
  GROUP BY winavg;
+--------+------------+
| winavg | AVG(plays) |
+--------+------------+
| 0.1765 |    17.0000 |
| 0.2500 |    26.0000 |
| 0.3636 |    22.0000 |
+--------+------------+
3 rows in set (0.00 sec)
```

You can filter on aggregate information using the `HAVING` clause. The `HAVING` clause is applied after `GROUP BY` and allows you to filter on aggregate data that is not available to the `WHERE` clause. Restrict the above example to results that involve an average number of plays over 20:

```
SELECT (wins / plays) AS winavg, AVG(plays) FROM plays
  GROUP BY winavg HAVING AVG(plays) > 20;
+--------+------------+
| winavg | AVG(plays) |
+--------+------------+
| 0.2500 |    26.0000 |
| 0.3636 |    22.0000 |
+--------+------------+
2 rows in set (0.00 sec)
```