15.2.13 SELECT Statement

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
    [HIGH_PRIORITY]
    [STRAIGHT_JOIN]
    [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
    [SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr] ...
    [into_option]
    [FROM table_references
      [PARTITION partition_list]]
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}, ... [WITH ROLLUP]]
    [HAVING where_condition]
    [WINDOW window_name AS (window_spec)
        [, window_name AS (window_spec)] ...]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [into_option]
    [FOR {UPDATE | SHARE}
        [OF tbl_name [, tbl_name] ...]
        [NOWAIT | SKIP LOCKED]
      | LOCK IN SHARE MODE]
    [into_option]

into_option: {
    INTO OUTFILE 'file_name'
        [CHARACTER SET charset_name]
        export_options
  | INTO DUMPFILE 'file_name'
  | INTO var_name [, var_name] ...
}

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

```
        [STARTING BY 'string']
        [TERMINATED BY 'string']
    ]
```

SELECT is used to retrieve rows selected from one or more tables, and can include UNION operations and subqueries. Beginning with MySQL 8.0.31, INTERSECT and EXCEPT operations are also supported. The UNION, INTERSECT, and EXCEPT operators are described in more detail later in this section. See also Section 15.2.15, “Subqueries”.

A SELECT statement can start with a WITH clause to define common table expressions accessible within the SELECT. See Section 15.2.20, “WITH (Common Table Expressions)”.

The most commonly used clauses of SELECT statements are these:

- Each ***select_expr*** indicates a column that you want to retrieve. There must be at least one ***select_expr***.
- ***table_references*** indicates the table or tables from which to retrieve rows. Its syntax is described in Section 15.2.13.2, “JOIN Clause”.
- SELECT supports explicit partition selection using the PARTITION clause with a list of partitions or subpartitions (or both) following the name of the table in a ***table_reference*** (see Section 15.2.13.2, “JOIN Clause”). In this case, rows are selected only from the partitions listed, and any other partitions of the table are ignored. For more information and examples, see Section 26.5, “Partition Selection”.
- 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 expression, you can use any of the functions and operators that MySQL supports, except for aggregate (group) functions. See Section 11.5, “Expressions”, and Chapter 14, *Functions and Operators*.

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

For example:

```
mysql> SELECT 1 + 1;
        -> 2
```

You are permitted to specify DUAL as a dummy table name in situations where no tables are referenced:

```
mysql> SELECT 1 + 1 FROM DUAL;
        -> 2
```

`DUAL` is purely for the convenience of people who require that all `SELECT` statements should have `FROM` and possibly other clauses. MySQL may ignore the clauses. MySQL does not require `FROM DUAL` if no tables are referenced.

In general, clauses used must be given in exactly the order shown in the syntax description. For example, a `HAVING` clause must come after any `GROUP BY` clause and before any `ORDER BY` clause. The `INTO` clause, if present, can appear in any position indicated by the syntax description, but within a given statement can appear only once, not in multiple positions. For more information about `INTO`, see Section 15.2.13.1, "SELECT ... INTO Statement".

The list of ***select_expr*** terms comprises the select list that indicates which columns to retrieve. Terms specify a column or expression or can use `*`-shorthand:

- A select list consisting only of a single unqualified `*` can be used as shorthand to select all columns from all tables:

  ```
  SELECT * FROM t1 INNER JOIN t2 ...
  ```

- ***tbl_name***`.*` can be used as a qualified shorthand to select all columns from the named table:

  ```
  SELECT t1.*, t2.* FROM t1 INNER JOIN t2 ...
  ```

- If a table has invisible columns, `*` and ***tbl_name***`.*` do not include them. To be included, invisible columns must be referenced explicitly.

- Use of an unqualified `*` with other items in the select list may produce a parse error. For example:

  ```
  SELECT id, * FROM t1
  ```

  To avoid this problem, use a qualified ***tbl_name***`.*` reference:

  ```
  SELECT id, t1.* FROM t1
  ```

Use qualified ***tbl_name*.*** references for each table in the select list:

```
SELECT AVG(score), t1.* FROM t1 ...
```

The following list provides additional information about other `SELECT` clauses:

- A ***select_expr*** can be given an alias using `AS` ***alias_name***. The alias is used as the expression's column name and can be used in `GROUP BY`, `ORDER BY`, or `HAVING` clauses. For example:

  ```
  SELECT CONCAT(last_name,', ',first_name) AS full_name
    FROM mytable ORDER BY full_name;
  ```

  The `AS` keyword is optional when aliasing a ***select_expr*** with an identifier. The preceding example could have been written like this:

  ```
  SELECT CONCAT(last_name,', ',first_name) full_name
    FROM mytable ORDER BY full_name;
  ```

  However, because the `AS` is optional, a subtle problem can occur if you forget the comma between two ***select_expr*** expressions: MySQL interprets the second as an alias name. For example, in the following statement, `columnb` is treated as an alias name:

  ```
  SELECT columna columnb FROM mytable;
  ```

  For this reason, it is good practice to be in the habit of using `AS` explicitly when specifying column aliases.

  It is not permissible to refer to a column alias in a `WHERE` clause, because the column value might not yet be determined when the `WHERE` clause is executed. See Section B.3.4.4, “Problems with Column Aliases”.

- The `FROM` ***table_references*** clause indicates the table or tables from which to retrieve rows. If you name more than one table, you are performing a join. For information on join syntax, see Section 15.2.13.2, “JOIN Clause”. For each table specified, you can optionally specify an alias.

  ```
  tbl_name [[AS] alias] [index_hint]
  ```

The use of index hints provides the optimizer with information about how to choose indexes during query processing. For a description of the syntax for specifying these hints, see Section 10.9.4, “Index Hints”.

You can use `SET max_seeks_for_key=`***`value`*** as an alternative way to force MySQL to prefer key scans instead of table scans. See Section 7.1.8, “Server System Variables”.

- You can refer to a table within the default database as ***`tbl_name`***, or as ***`db_name`***.***`tbl_name`*** to specify a database explicitly. You can refer to a column as ***`col_name`***, ***`tbl_name`***.***`col_name`***, or ***`db_name`***.***`tbl_name`***.***`col_name`***. You need not specify a ***`tbl_name`*** or ***`db_name`***.***`tbl_name`*** prefix for a column reference unless the reference would be ambiguous. See Section 11.2.2, “Identifier Qualifiers”, for examples of ambiguity that require the more explicit column reference forms.

- A table reference can be aliased using ***`tbl_name`*** `AS` ***`alias_name`*** or ***`tbl_name alias_name`***. These statements are equivalent:

```
SELECT t1.name, t2.salary FROM employee AS t1, info AS t2
  WHERE t1.name = t2.name;

SELECT t1.name, t2.salary FROM employee t1, info t2
  WHERE t1.name = t2.name;
```

- Columns selected for output can be referred to in `ORDER BY` and `GROUP BY` clauses using column names, column aliases, or column positions. Column positions are integers and begin with 1:

```
SELECT college, region, seed FROM tournament
  ORDER BY region, seed;

SELECT college, region AS r, seed AS s FROM tournament
  ORDER BY r, s;

SELECT college, region, seed FROM tournament
  ORDER BY 2, 3;
```

To sort in reverse order, add the `DESC` (descending) keyword to the name of the column in the `ORDER BY` clause that you are sorting by. The default is ascending order; this can be specified explicitly using the `ASC` keyword.

If `ORDER BY` occurs within a parenthesized query expression and also is applied in the outer query, the results are undefined and may change in a future version of MySQL.

Use of column positions is deprecated because the syntax has been removed from the SQL standard.

- Prior to MySQL 8.0.13, MySQL supported a nonstandard syntax extension that permitted explicit `ASC` or `DESC` designators for `GROUP BY` columns. MySQL 8.0.12 and later supports `ORDER BY` with grouping functions so that use of this extension is no longer necessary. (Bug #86312, Bug #26073525) This also means you can sort on an arbitrary column or columns when using `GROUP BY`, like this:

  ```
  SELECT a, b, COUNT(c) AS t FROM test_table GROUP BY a,b ORDER BY a,t DESC;
  ```

  As of MySQL 8.0.13, the `GROUP BY` extension is no longer supported: `ASC` or `DESC` designators for `GROUP BY` columns are not permitted.

- When you use `ORDER BY` or `GROUP BY` to sort a column in a `SELECT`, the server sorts values using only the initial number of bytes indicated by the `max_sort_length` system variable.

- MySQL extends the use of `GROUP BY` to permit selecting fields that are not mentioned in the `GROUP BY` clause. If you are not getting the results that you expect from your query, please read the description of `GROUP BY` found in Section 14.19, "Aggregate Functions".

- `GROUP BY` permits a `WITH ROLLUP` modifier. See Section 14.19.2, "GROUP BY Modifiers".

  Previously, it was not permitted to use `ORDER BY` in a query having a `WITH ROLLUP` modifier. This restriction is lifted as of MySQL 8.0.12. See Section 14.19.2, "GROUP BY Modifiers".

- The `HAVING` clause, like the `WHERE` clause, specifies selection conditions. The `WHERE` clause specifies conditions on columns in the select list, but cannot refer to aggregate functions. The `HAVING` clause specifies conditions on groups, typically formed by the `GROUP BY` clause. The query result includes only groups satisfying the `HAVING` conditions. (If no `GROUP BY` is present, all rows implicitly form a single aggregate group.)

  The `HAVING` clause is applied nearly last, just before items are sent to the client, with no optimization. (`LIMIT` is applied after `HAVING`.)

  The SQL standard requires that `HAVING` must reference only columns in the `GROUP BY` clause or columns used in aggregate functions. However, MySQL supports an extension to this behavior, and permits `HAVING` to refer to columns in the `SELECT` list and columns in outer subqueries as well.

  If the `HAVING` clause refers to a column that is ambiguous, a warning occurs. In the following statement, `col2` is ambiguous because it is used as both an alias and a column name:

```
SELECT COUNT(col1) AS col2 FROM t GROUP BY col2 HAVING col2 = 2;
```

  Preference is given to standard SQL behavior, so if a `HAVING` column name is used both in `GROUP BY` and as an aliased column in the select column list, preference is given to the column in the `GROUP BY` column.

- Do not use `HAVING` for items that should be in the `WHERE` clause. For example, do not write the following:

  ```
  SELECT col_name FROM tbl_name HAVING col_name > 0;
  ```

  Write this instead:

  ```
  SELECT col_name FROM tbl_name WHERE col_name > 0;
  ```

- The `HAVING` clause can refer to aggregate functions, which the `WHERE` clause cannot:

  ```
  SELECT user, MAX(salary) FROM users
    GROUP BY user HAVING MAX(salary) > 10;
  ```

  (This did not work in some older versions of MySQL.)

- MySQL permits duplicate column names. That is, there can be more than one ***`select_expr`*** with the same name. This is an extension to standard SQL. Because MySQL also permits `GROUP BY` and `HAVING` to refer to ***`select_expr`*** values, this can result in an ambiguity:

  ```
  SELECT 12 AS a, a FROM t GROUP BY a;
  ```

  In that statement, both columns have the name `a`. To ensure that the correct column is used for grouping, use different names for each ***`select_expr`***.

- The `WINDOW` clause, if present, defines named windows that can be referred to by window functions. For details, see Section 14.20.4, “Named Windows”.

- MySQL resolves unqualified column or alias references in `ORDER BY` clauses by searching in the ***`select_expr`*** values, then in the columns of the tables in the `FROM` clause. For `GROUP BY` or `HAVING` clauses, it searches the FROM clause before searching in the ***select_expr*** values. (For GROUP BY and HAVING, this differs from the pre-MySQL 5.0 behavior that used the same rules as for ORDER BY.)

- The LIMIT clause can be used to constrain the number of rows returned by the SELECT statement. LIMIT takes one or two numeric arguments, which must both be nonnegative integer constants, with these exceptions:

  - Within prepared statements, LIMIT parameters can be specified using ? placeholder markers.

  - Within stored programs, LIMIT parameters can be specified using integer-valued routine parameters or local variables.

  With two arguments, the first argument specifies the offset of the first row to return, and the second specifies the maximum number of rows to return. The offset of the initial row is 0 (not 1):

```
SELECT * FROM tbl LIMIT 5,10;  # Retrieve rows 6-15
```

  To retrieve all rows from a certain offset up to the end of the result set, you can use some large number for the second parameter. This statement retrieves all rows from the 96th row to the last:

```
SELECT * FROM tbl LIMIT 95,18446744073709551615;
```

  With one argument, the value specifies the number of rows to return from the beginning of the result set:

```
SELECT * FROM tbl LIMIT 5;     # Retrieve first 5 rows
```

  In other words, LIMIT ***row_count*** is equivalent to LIMIT 0, ***row_count***.

  For prepared statements, you can use placeholders. The following statements return one row from the tbl table:

```
SET @a=1;
PREPARE STMT FROM 'SELECT * FROM tbl LIMIT ?';
EXECUTE STMT USING @a;
```

  The following statements return the second to sixth rows from the tbl table:

```
SET @skip=1; SET @numrows=5;
PREPARE STMT FROM 'SELECT * FROM tbl LIMIT ?, ?';
EXECUTE STMT USING @skip, @numrows;
```

For compatibility with PostgreSQL, MySQL also supports the `LIMIT` ***`row_count`*** `OFFSET` ***`offset`*** syntax.

If `LIMIT` occurs within a parenthesized query expression and also is applied in the outer query, the results are undefined and may change in a future version of MySQL.

- The `SELECT ... INTO` form of `SELECT` enables the query result to be written to a file or stored in variables. For more information, see Section 15.2.13.1, "SELECT ... INTO Statement".

- If you use `FOR UPDATE` with a storage engine that uses page or row locks, rows examined by the query are write-locked until the end of the current transaction.

  You cannot use `FOR UPDATE` as part of the `SELECT` in a statement such as `CREATE TABLE` ***`new_table`*** `SELECT ... FROM` ***`old_table`*** `...`. (If you attempt to do so, the statement is rejected with the error `Can't update table '`***`old_table`***`' while '`***`new_table`***`' is being created`.)

  `FOR SHARE` and `LOCK IN SHARE MODE` set shared locks that permit other transactions to read the examined rows but not to update or delete them. `FOR SHARE` and `LOCK IN SHARE MODE` are equivalent. However, `FOR SHARE`, like `FOR UPDATE`, supports `NOWAIT`, `SKIP LOCKED`, and `OF` ***`tbl_name`*** options. `FOR SHARE` is a replacement for `LOCK IN SHARE MODE`, but `LOCK IN SHARE MODE` remains available for backward compatibility.

  `NOWAIT` causes a `FOR UPDATE` or `FOR SHARE` query to execute immediately, returning an error if a row lock cannot be obtained due to a lock held by another transaction.

  `SKIP LOCKED` causes a `FOR UPDATE` or `FOR SHARE` query to execute immediately, excluding rows from the result set that are locked by another transaction.

  `NOWAIT` and `SKIP LOCKED` options are unsafe for statement-based replication.

  > **Note**
  >
  > Queries that skip locked rows return an inconsistent view of the data. `SKIP LOCKED` is therefore not suitable for general transactional work. However, it may be used to avoid lock contention when multiple sessions access the same queue-like table.

OF ***tbl_name*** applies FOR UPDATE and FOR SHARE queries to named tables. For example:

```
SELECT * FROM t1, t2 FOR SHARE OF t1 FOR UPDATE OF t2;
```

All tables referenced by the query block are locked when OF ***tbl_name*** is omitted. Consequently, using a locking clause without OF ***tbl_name*** in combination with another locking clause returns an error. Specifying the same table in multiple locking clauses returns an error. If an alias is specified as the table name in the SELECT statement, a locking clause may only use the alias. If the SELECT statement does not specify an alias explicitly, the locking clause may only specify the actual table name.

For more information about FOR UPDATE and FOR SHARE, see Section 17.7.2.4, "Locking Reads". For additional information about NOWAIT and SKIP LOCKED options, see Locking Read Concurrency with NOWAIT and SKIP LOCKED.

Following the SELECT keyword, you can use a number of modifiers that affect the operation of the statement. HIGH_PRIORITY, STRAIGHT_JOIN, and modifiers beginning with SQL_ are MySQL extensions to standard SQL.

- The ALL and DISTINCT modifiers specify whether duplicate rows should be returned. ALL (the default) specifies that all matching rows should be returned, including duplicates. DISTINCT specifies removal of duplicate rows from the result set. It is an error to specify both modifiers. DISTINCTROW is a synonym for DISTINCT.

  In MySQL 8.0.12 and later, DISTINCT can be used with a query that also uses WITH ROLLUP. (Bug #87450, Bug #26640100)

- HIGH_PRIORITY gives the SELECT higher priority than a statement that updates a table. You should use this only for queries that are very fast and must be done at once. A SELECT HIGH_PRIORITY query that is issued while the table is locked for reading runs even if there is an update statement waiting for the table to be free. This affects only storage engines that use only table-level locking (such as MyISAM, MEMORY, and MERGE).

  HIGH_PRIORITY cannot be used with SELECT statements that are part of a UNION.

- STRAIGHT_JOIN forces the optimizer to join the tables in the order in which they are listed in the FROM clause. You can use this to speed up a query if the optimizer joins the tables in nonoptimal order. STRAIGHT_JOIN also can be used in the ***table_references*** list. See Section 15.2.13.2, "JOIN Clause".

STRAIGHT_JOIN does not apply to any table that the optimizer treats as a const or system table. Such a table produces a single row, is read during the optimization phase of query execution, and references to its columns are replaced with the appropriate column values before query execution proceeds. These tables appear first in the query plan displayed by EXPLAIN. See Section 10.8.1, “Optimizing Queries with EXPLAIN”. This exception may not apply to const or system tables that are used on the NULL-complemented side of an outer join (that is, the right-side table of a LEFT JOIN or the left-side table of a RIGHT JOIN.

- SQL_BIG_RESULT or SQL_SMALL_RESULT can be used with GROUP BY or DISTINCT to tell the optimizer that the result set has many rows or is small, respectively. For SQL_BIG_RESULT, MySQL directly uses disk-based temporary tables if they are created, and prefers sorting to using a temporary table with a key on the GROUP BY elements. For SQL_SMALL_RESULT, MySQL uses in-memory temporary tables to store the resulting table instead of using sorting. This should not normally be needed.

- SQL_BUFFER_RESULT forces the result to be put into a temporary table. This helps MySQL free the table locks early and helps in cases where it takes a long time to send the result set to the client. This modifier can be used only for top-level SELECT statements, not for subqueries or following UNION.

- SQL_CALC_FOUND_ROWS tells MySQL to calculate how many rows there would be in the result set, disregarding any LIMIT clause. The number of rows can then be retrieved with SELECT FOUND_ROWS(). See Section 14.15, “Information Functions”.

  > **Note**
  >
  > The SQL_CALC_FOUND_ROWS query modifier and accompanying FOUND_ROWS() function are deprecated as of MySQL 8.0.17; expect them to be removed in a future version of MySQL. See the FOUND_ROWS() description for information about an alternative strategy.

- The SQL_CACHE and SQL_NO_CACHE modifiers were used with the query cache prior to MySQL 8.0. The query cache was removed in MySQL 8.0. The SQL_CACHE modifier was removed as well. SQL_NO_CACHE is deprecated, and has no effect; expect it to be removed in a future MySQL release.