

13.1.7. SELECT Syntax

[13.1.7.1. JOIN Syntax](#)

[13.1.7.2. UNION 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, ...
  [INTO OUTFILE 'file_name' export_options
   | INTO DUMPFILE 'file_name']
  [FROM table_references
   [WHERE where_definition]
   [GROUP BY {col_name | expr | position}
    [ASC | DESC], ... [WITH ROLLUP]]
   [HAVING where_definition]
   [ORDER BY {col_name | expr | position}
    [ASC | DESC] , ...]
   [LIMIT {[offset,] row_count | row_count OFFSET offset}]
   [PROCEDURE procedure_name(argument_list)]
   [FOR UPDATE | LOCK IN SHARE MODE]]

```

SELECT is used to retrieve rows selected from one or more tables. Support for UNION statements and subqueries is available as of MySQL 4.0 and 4.1, respectively. See [Section 13.1.7.2, "UNION Syntax"](#) and [Section 13.1.8, "Subquery Syntax"](#).

- Each *select_expr* indicates a column you want to retrieve.
- *table_references* indicates the table or tables from which to retrieve rows. Its syntax is described in [Section 13.1.7.1, "JOIN Syntax"](#).
- *where_definition* consists of the keyword WHERE followed by an expression that indicates the condition or conditions that rows must satisfy to be selected.

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

For example:

```

mysql> SELECT 1 + 1;
-> 2

```

All 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.

- 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:

```

mysql> 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`. The preceding example could have been written like this:

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

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:

```
mysql> SELECT columna columnb FROM mytable;
```

- It is not allowable to use a column alias in a `WHERE` clause, because the column value might not yet be determined when the `WHERE` clause is executed. See [Section A.5.4, “Problems with Column Aliases”](#).
- The `FROM table_references` clause indicates the 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 13.1.7.1, “JOIN Syntax”](#). For each table specified, you can optionally specify an alias.

```
tbl_name [[AS] alias]
        [[USE INDEX (key_list)]
         | [IGNORE INDEX (key_list)]
         | [FORCE INDEX (key_list)]]
```

The use of `USE INDEX`, `IGNORE INDEX`, `FORCE INDEX` to give the optimizer hints about how to choose indexes is described in [Section 13.1.7.1, “JOIN Syntax”](#).

In MySQL 4.0.14, you can use `SET max_seeks_for_key=value` as an alternative way to force MySQL to prefer key scans instead of table scans.

- You can refer to a table within the current database as `tbl_name` (within the current database), or as `db_name.tbl_name` to explicitly specify a database. 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 9.2, “Database, Table, Index, Column, and Alias Names”](#) for examples of ambiguity that require the more explicit column reference forms.
- From MySQL 4.1.0 on, you are allowed 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 a compatibility feature. Some other servers require this syntax.

- A table reference can be aliased using `tbl_name AS alias_name` or `tbl_name alias_name`:

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

- In the `WHERE` clause, you can use any of the functions that MySQL supports, except for aggregate (summary) functions. See [Chapter 12, Functions and Operators](#).
- 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:

```
mysql> SELECT college, region, seed FROM tournament
-> ORDER BY region, seed;
mysql> SELECT college, region AS r, seed AS s FROM tournament
-> ORDER BY r, s;
mysql> 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.

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

- If you use `GROUP BY`, output rows are sorted according to the `GROUP BY` columns as if you had an `ORDER BY` for the same columns. MySQL has extended the `GROUP BY` clause as of version 3.23.34 so that you can also specify `ASC` and `DESC` after columns named in the clause:

```
SELECT a, COUNT(b) FROM test_table GROUP BY a DESC
```

- MySQL extends the use of `GROUP BY` to allow you to select fields that are not mentioned in the `GROUP BY` clause. If you are not getting the results you expect from your query, please read the `GROUP BY` description. See [Section 12.9, "Functions and Modifiers for Use with GROUP BY Clauses"](#).
- As of MySQL 4.1.1, `GROUP BY` allows a `WITH ROLLUP` modifier. See [Section 12.9.2, "GROUP BY Modifiers"](#).
- The `HAVING` clause is applied nearly last, just before items are sent to the client, with no optimization. (`LIMIT` is applied after `HAVING`.)

Before MySQL 5.0.2, a `HAVING` clause can refer to any column or alias named in a `select_expr` in the `SELECT` list or in outer subqueries, and to aggregate functions. Standard SQL requires that `HAVING` must reference only columns in the `GROUP BY` clause or columns used in aggregate functions. To accommodate both standard SQL and the MySQL-specific behavior of being able to refer columns in the `SELECT` list, MySQL 5.0.2 and up allows `HAVING` to refer to columns in the `SELECT` list, columns in the `GROUP BY` clause, columns in outer subqueries, and to aggregate functions.

For example, the following statement works in MySQL 5.0.2 but produces an error for

earlier versions:

```
mysql> SELECT COUNT(*) FROM t GROUP BY col1 HAVING col1 = 2;
```

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 both as an alias and as a column name:

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

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

- Don't use `HAVING` for items that should be in the `WHERE` clause. For example, do not write this:

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

Write this instead:

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

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

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

However, that does not work in older MySQL servers (before version 3.22.5). Instead, you can use a column alias in the select list and refer to the alias in the `HAVING` clause:

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

- 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 be integer constants.

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):

```
mysql> SELECT * FROM table LIMIT 5,10; # Retrieve rows 6-15
```

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

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:

```
mysql> SELECT * FROM table LIMIT 95,18446744073709551615;
```

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

```
mysql> SELECT * FROM table LIMIT 5;      # Retrieve first 5 rows
```

In other words, `LIMIT n` is equivalent to `LIMIT 0,n`.

- The `SELECT ... INTO OUTFILE 'file_name'` form of `SELECT` writes the selected rows to a file. The file is created on the server host, so you must have the `FILE` privilege to use this syntax. The file cannot currently exist, which among other things prevents files such as `/etc/passwd` and database tables from being destroyed.

The `SELECT ... INTO OUTFILE` statement is intended primarily to let you very quickly dump a table on the server machine. If you want to create the resulting file on some client host other than the server host, you can't use `SELECT ... INTO OUTFILE`. In that case, you should instead use some command like `mysql -e "SELECT ..." > file_name` on the client host to generate the file.

`SELECT ... INTO OUTFILE` is the complement of `LOAD DATA INFILE`; the syntax for the `export_options` part of the statement consists of the same `FIELDS` and `LINES` clauses that are used with the `LOAD DATA INFILE` statement. See [Section 13.1.5, "LOAD DATA INFILE Syntax"](#).

`FIELDS ESCAPED BY` controls how to write special characters. If the `FIELDS ESCAPED BY` character is not empty, it is used to prefix the following characters on output:

- The `FIELDS ESCAPED BY` character
- The `FIELDS [OPTIONALLY] ENCLOSED BY` character
- The first character of the `FIELDS TERMINATED BY` and `LINES TERMINATED BY` values
- ASCII 0 (what is actually written following the escape character is ASCII '0', not a zero-valued byte)

If the `FIELDS ESCAPED BY` character is empty, no characters are escaped and `NULL` is output as `NULL`, not `\N`. It is probably not a good idea to specify an empty escape character, particularly if field values in your data contain any of the characters in the list just given.

The reason for the above is that you *must* escape any `FIELDS TERMINATED BY`, `ENCLOSED BY`, `ESCAPED BY`, or `LINES TERMINATED BY` characters to reliably be able to read the file back. ASCII NUL is escaped to make it easier to view with some pagers.

The resulting file doesn't have to conform to SQL syntax, so nothing else need be escaped.

Here is an example that produces a file in the comma-separated values format used by many programs:

```
SELECT a,b,a+b INTO OUTFILE '/tmp/result.text'  
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY ''''  
LINES TERMINATED BY '\n'  
FROM test_table;
```

- If you use `INTO DUMPFILE` instead of `INTO OUTFILE`, MySQL writes only one row into the file, without any column or line termination and without performing any escape processing. This is useful if you want to store a BLOB value in a file.
- **Note:** Any file created by `INTO OUTFILE` or `INTO DUMPFILE` is writable by all users on the server host. The reason for this is that the MySQL server can't create a file that is owned by anyone other than the user it's running as (you should never run `mysqld` as `root`). The file thus must be world-writable so that you can manipulate its contents.
- A `PROCEDURE` clause names a procedure that should process the data in the result set. For an example, see [Section 27.3.1, "Procedure Analyse"](#).
- If you use `FOR UPDATE` on a storage engine that uses page or row locks, rows examined by the query are write-locked until the end of the current transaction. Using `LOCK IN SHARE MODE` sets a shared lock that prevents other transactions from updating or deleting the examined rows. See [Section 15.11.5, "Locking Reads SELECT ... FOR UPDATE and SELECT ... LOCK IN SHARE MODE"](#).

Following the `SELECT` keyword, you can give a number of options that affect the operation of the statement.

The `ALL`, `DISTINCT`, and `DISTINCTROW` options specify whether duplicate rows should be returned. If none of these options are given, the default is `ALL` (all matching rows are returned). `DISTINCT` and `DISTINCTROW` are synonyms and specify that duplicate rows in the result set should be removed.

`HIGH_PRIORITY`, `STRAIGHT_JOIN`, and options beginning with `SQL_` are MySQL extensions to standard SQL.

- `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.

`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 non-optimal order. See [Section 7.2.1, "EXPLAIN Syntax \(Get Information About a SELECT\)"](#). `STRAIGHT_JOIN` also can be used in the `table_references` list. See [Section 13.1.7.1, "JOIN Syntax"](#).
- `SQL_BIG_RESULT` can be used with `GROUP BY` or `DISTINCT` to tell the optimizer that the result set has many rows. In this case, MySQL directly uses disk-based temporary

tables if needed. MySQL also, in this case, prefers sorting to using a temporary table with a key on the `GROUP BY` elements.

- `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.
- `SQL_SMALL_RESULT` can be used with `GROUP BY` or `DISTINCT` to tell the optimizer that the result set is small. In this case, MySQL uses fast temporary tables to store the resulting table instead of using sorting. In MySQL 3.23 and up, this shouldn't normally be needed.
- `SQL_CALC_FOUND_ROWS` (available in MySQL 4.0.0 and up) 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 12.8.3, "Information Functions"](#).

Before MySQL 4.1.0, this option does not work with `LIMIT 0`, which is optimized to return instantly (resulting in a row count of 0). See [Section 7.2.12, "How MySQL Optimizes LIMIT"](#).

- `SQL_CACHE` tells MySQL to store the query result in the query cache if you are using a `query_cache_type` value of 2 or `DEMAND`. For a query that uses `UNION` or subqueries, this option takes effect to be used in any `SELECT` of the query. See [Section 5.12, "The MySQL Query Cache"](#).
- `SQL_NO_CACHE` tells MySQL not to store the query result in the query cache. See [Section 5.12, "The MySQL Query Cache"](#). For a query that uses `UNION` or subqueries, this option takes effect to be used in any `SELECT` of the query.