```
+-----------------------------+
mysql> SELECT ST_AsText(ST_Validate(@ls2));
+-----------------------------+
| ST_AsText(ST_Validate(@ls2)) |
+-----------------------------+
| LINESTRING(0 0,1 1)         |
+-----------------------------+
```

# 12.18 JSON Functions

The functions described in this section perform operations on JSON values. For discussion of the `JSON` data type and additional examples showing how to use these functions, see Section 11.5, "The JSON Data Type".

For functions that take a JSON argument, an error occurs if the argument is not a valid JSON value. Arguments parsed as JSON are indicated by *`json_doc`*; arguments indicated by *`val`* are not parsed.

Functions that return JSON values always perform normalization of these values (see Normalization, Merging, and Autowrapping of JSON Values), and thus orders them. *The precise outcome of the sort is subject to change at any time; do not rely on it to be consistent between releases*.

A set of spatial functions for operating on GeoJSON values is also available. See Section 12.17.11, "Spatial GeoJSON Functions".

## 12.18.1 JSON Function Reference

**Table 12.22 JSON Functions**

| Name | Description | Introduced | Deprecated |
|---|---|---|---|
| `->` | Return value from JSON column after evaluating path; equivalent to JSON_EXTRACT(). | | |
| `->>` | Return value from JSON column after evaluating path and unquoting the result; equivalent to JSON_UNQUOTE(JSON_EXTRACT()). | | |
| `JSON_ARRAY()` | Create JSON array | | |
| `JSON_ARRAY_APPEND()` | Append data to JSON document | | |
| `JSON_ARRAY_INSERT()` | Insert into JSON array | | |
| `JSON_CONTAINS()` | Whether JSON document contains specific object at path | | |
| `JSON_CONTAINS_PATH()` | Whether JSON document contains any data at path | | |
| `JSON_DEPTH()` | Maximum depth of JSON document | | |
| `JSON_EXTRACT()` | Return data from JSON document | | |
| `JSON_INSERT()` | Insert data into JSON document | | |
| `JSON_KEYS()` | Array of keys from JSON document | | |

| Name | Description | Introduced | Deprecated |
|---|---|---|---|
| JSON_LENGTH() | Number of elements in JSON document | | |
| JSON_MERGE() | Merge JSON documents, preserving duplicate keys. Deprecated synonym for JSON_MERGE_PRESERVE() | | Yes |
| JSON_MERGE_PATCH() | Merge JSON documents, replacing values of duplicate keys | | |
| JSON_MERGE_PRESERVE() | Merge JSON documents, preserving duplicate keys | | |
| JSON_OBJECT() | Create JSON object | | |
| JSON_OVERLAPS() | Compares two JSON documents, returns TRUE (1) if these have any key-value pairs or array elements in common, otherwise FALSE (0) | 8.0.17 | |
| JSON_PRETTY() | Print a JSON document in human-readable format | | |
| JSON_QUOTE() | Quote JSON document | | |
| JSON_REMOVE() | Remove data from JSON document | | |
| JSON_REPLACE() | Replace values in JSON document | | |
| JSON_SCHEMA_VALID() | Validate JSON document against JSON schema; returns TRUE/1 if document validates against schema, or FALSE/0 if it does not | 8.0.17 | |
| JSON_SCHEMA_VALIDATION_REPORT() | Validate JSON document against JSON schema; returns report in JSON format on outcome on validation including success or failure and reasons for failure | 8.0.17 | |
| JSON_SEARCH() | Path to value within JSON document | | |
| JSON_SET() | Insert data into JSON document | | |
| JSON_STORAGE_FREE() | Freed space within binary representation | | |

| Name | Description | Introduced | Deprecated |
|---|---|---|---|
| | of JSON column value following partial update | | |
| JSON_STORAGE_SIZE() | Space used for storage of binary representation of a JSON document | | |
| JSON_TABLE() | Return data from a JSON expression as a relational table | | |
| JSON_TYPE() | Type of JSON value | | |
| JSON_UNQUOTE() | Unquote JSON value | | |
| JSON_VALID() | Whether JSON value is valid | | |
| JSON_VALUE() | Extract value from JSON document at location pointed to by path provided; return this value as VARCHAR(512) or specified type | 8.0.21 | |
| MEMBER OF() | Returns true (1) if first operand matches any element of JSON array passed as second operand, otherwise returns false (0) | 8.0.17 | |

MySQL supports two aggregate JSON functions JSON_ARRAYAGG() and JSON_OBJECTAGG(). See Section 12.20, “Aggregate Functions”, for descriptions of these.

MySQL also supports “pretty-printing” of JSON values in an easy-to-read format, using the JSON_PRETTY() function. You can see how much storage space a given JSON value takes up, and how much space remains for additional storage, using JSON_STORAGE_SIZE() and JSON_STORAGE_FREE(), respectively. For complete descriptions of these functions, see Section 12.18.8, “JSON Utility Functions”.

## 12.18.2 Functions That Create JSON Values

The functions listed in this section compose JSON values from component elements.

- JSON_ARRAY([*val*[, *val*] ...])

  Evaluates a (possibly empty) list of values and returns a JSON array containing those values.

```
mysql> SELECT JSON_ARRAY(1, "abc", NULL, TRUE, CURTIME());
+---------------------------------------------+
| JSON_ARRAY(1, "abc", NULL, TRUE, CURTIME()) |
+---------------------------------------------+
| [1, "abc", null, true, "11:30:24.000000"]   |
+---------------------------------------------+
```

- JSON_OBJECT([*key*, *val*[, *key*, *val*] ...])

  Evaluates a (possibly empty) list of key-value pairs and returns a JSON object containing those pairs. An error occurs if any key name is NULL or the number of arguments is odd.

```
mysql> SELECT JSON_OBJECT('id', 87, 'name', 'carrot');
+-----------------------------------+
| JSON_OBJECT('id', 87, 'name', 'carrot') |
```

```
+----------------------------------------+
| {"id": 87, "name": "carrot"}           |
+----------------------------------------+
```

- `JSON_QUOTE(string)`

  Quotes a string as a JSON value by wrapping it with double quote characters and escaping interior quote and other characters, then returning the result as a `utf8mb4` string. Returns `NULL` if the argument is `NULL`.

  This function is typically used to produce a valid JSON string literal for inclusion within a JSON document.

  Certain special characters are escaped with backslashes per the escape sequences shown in Table 12.23, “JSON_UNQUOTE() Special Character Escape Sequences”.

```
mysql> SELECT JSON_QUOTE('null'), JSON_QUOTE('"null"');
+-------------------+---------------------+
| JSON_QUOTE('null') | JSON_QUOTE('"null"') |
+-------------------+---------------------+
| "null"             | "\"null\""           |
+-------------------+---------------------+
mysql> SELECT JSON_QUOTE('[1, 2, 3]');
+-------------------------+
| JSON_QUOTE('[1, 2, 3]') |
+-------------------------+
| "[1, 2, 3]"             |
+-------------------------+
```

You can also obtain JSON values by casting values of other types to the `JSON` type using `CAST(value AS JSON)`; see Converting between JSON and non-JSON values, for more information.

Two aggregate functions generating JSON values are available. `JSON_ARRAYAGG()` returns a result set as a single JSON array, and `JSON_OBJECTAGG()` returns a result set as a single JSON object. For more information, see Section 12.20, “Aggregate Functions”.

## 12.18.3 Functions That Search JSON Values

The functions in this section perform search or comparison operations on JSON values to extract data from them, report whether data exists at a location within them, or report the path to data within them. The `MEMBER OF()` operator is also documented herein.

- `JSON_CONTAINS(target, candidate[, path])`

  Indicates by returning 1 or 0 whether a given *`candidate`* JSON document is contained within a *`target`* JSON document, or—if a *`path`* argument was supplied—whether the candidate is found at a specific path within the target. Returns `NULL` if any argument is `NULL`, or if the path argument does not identify a section of the target document. An error occurs if *`target`* or *`candidate`* is not a valid JSON document, or if the *`path`* argument is not a valid path expression or contains a `*` or `**` wildcard.

  To check only whether any data exists at the path, use `JSON_CONTAINS_PATH()` instead.

  The following rules define containment:

  - A candidate scalar is contained in a target scalar if and only if they are comparable and are equal. Two scalar values are comparable if they have the same `JSON_TYPE()` types, with the exception that values of types `INTEGER` and `DECIMAL` are also comparable to each other.
  - A candidate array is contained in a target array if and only if every element in the candidate is contained in some element of the target.
  - A candidate nonarray is contained in a target array if and only if the candidate is contained in some element of the target.

- A candidate object is contained in a target object if and only if for each key in the candidate there is a key with the same name in the target and the value associated with the candidate key is contained in the value associated with the target key.

Otherwise, the candidate value is not contained in the target document.

Starting with MySQL 8.0.17, queries using `JSON_CONTAINS()` on `InnoDB` tables can be optimized using multi-valued indexes; see Multi-Valued Indexes, for more information.

```
mysql> SET @j = '{"a": 1, "b": 2, "c": {"d": 4}}';
mysql> SET @j2 = '1';
mysql> SELECT JSON_CONTAINS(@j, @j2, '$.a');
+-------------------------------+
| JSON_CONTAINS(@j, @j2, '$.a') |
+-------------------------------+
|                             1 |
+-------------------------------+
mysql> SELECT JSON_CONTAINS(@j, @j2, '$.b');
+-------------------------------+
| JSON_CONTAINS(@j, @j2, '$.b') |
+-------------------------------+
|                             0 |
+-------------------------------+

mysql> SET @j2 = '{"d": 4}';
mysql> SELECT JSON_CONTAINS(@j, @j2, '$.a');
+-------------------------------+
| JSON_CONTAINS(@j, @j2, '$.a') |
+-------------------------------+
|                             0 |
+-------------------------------+
mysql> SELECT JSON_CONTAINS(@j, @j2, '$.c');
+-------------------------------+
| JSON_CONTAINS(@j, @j2, '$.c') |
+-------------------------------+
|                             1 |
+-------------------------------+
```

- `JSON_CONTAINS_PATH(json_doc, one_or_all, path[, path] ...)`

  Returns 0 or 1 to indicate whether a JSON document contains data at a given path or paths. Returns `NULL` if any argument is `NULL`. An error occurs if the *`json_doc`* argument is not a valid JSON document, any *`path`* argument is not a valid path expression, or *`one_or_all`* is not `'one'` or `'all'`.

  To check for a specific value at a path, use `JSON_CONTAINS()` instead.

  The return value is 0 if no specified path exists within the document. Otherwise, the return value depends on the *`one_or_all`* argument:

  - `'one'`: 1 if at least one path exists within the document, 0 otherwise.
  - `'all'`: 1 if all paths exist within the document, 0 otherwise.

```
mysql> SET @j = '{"a": 1, "b": 2, "c": {"d": 4}}';
mysql> SELECT JSON_CONTAINS_PATH(@j, 'one', '$.a', '$.e');
+-------------------------------------------+
| JSON_CONTAINS_PATH(@j, 'one', '$.a', '$.e') |
+-------------------------------------------+
|                                         1 |
+-------------------------------------------+
mysql> SELECT JSON_CONTAINS_PATH(@j, 'all', '$.a', '$.e');
+-------------------------------------------+
| JSON_CONTAINS_PATH(@j, 'all', '$.a', '$.e') |
+-------------------------------------------+
|                                         0 |
+-------------------------------------------+
```

```
mysql> SELECT JSON_CONTAINS_PATH(@j, 'one', '$.c.d');
+-----------------------------------------+
| JSON_CONTAINS_PATH(@j, 'one', '$.c.d') |
+-----------------------------------------+
|                                       1 |
+-----------------------------------------+
mysql> SELECT JSON_CONTAINS_PATH(@j, 'one', '$.a.d');
+-----------------------------------------+
| JSON_CONTAINS_PATH(@j, 'one', '$.a.d') |
+-----------------------------------------+
|                                       0 |
+-----------------------------------------+
```

- `JSON_EXTRACT(json_doc, path[, path] ...)`

  Returns data from a JSON document, selected from the parts of the document matched by the *`path`* arguments. Returns `NULL` if any argument is `NULL` or no paths locate a value in the document. An error occurs if the *`json_doc`* argument is not a valid JSON document or any *`path`* argument is not a valid path expression.

  The return value consists of all values matched by the *`path`* arguments. If it is possible that those arguments could return multiple values, the matched values are autowrapped as an array, in the order corresponding to the paths that produced them. Otherwise, the return value is the single matched value.

```
mysql> SELECT JSON_EXTRACT('[10, 20, [30, 40]]', '$[1]');
+-------------------------------------------+
| JSON_EXTRACT('[10, 20, [30, 40]]', '$[1]') |
+-------------------------------------------+
| 20                                        |
+-------------------------------------------+
mysql> SELECT JSON_EXTRACT('[10, 20, [30, 40]]', '$[1]', '$[0]');
+-----------------------------------------------------+
| JSON_EXTRACT('[10, 20, [30, 40]]', '$[1]', '$[0]') |
+-----------------------------------------------------+
| [20, 10]                                            |
+-----------------------------------------------------+
mysql> SELECT JSON_EXTRACT('[10, 20, [30, 40]]', '$[2][*]');
+----------------------------------------------+
| JSON_EXTRACT('[10, 20, [30, 40]]', '$[2][*]') |
+----------------------------------------------+
| [30, 40]                                     |
+----------------------------------------------+
```

  MySQL supports the `->` operator as shorthand for this function as used with 2 arguments where the left hand side is a `JSON` column identifier (not an expression) and the right hand side is the JSON path to be matched within the column.

- *`column->path`*

  The `->` operator serves as an alias for the `JSON_EXTRACT()` function when used with two arguments, a column identifier on the left and a JSON path (a string literal) on the right that is evaluated against the JSON document (the column value). You can use such expressions in place of column references wherever they occur in SQL statements.

  The two `SELECT` statements shown here produce the same output:

```
mysql> SELECT c, JSON_EXTRACT(c, "$.id"), g
    > FROM jemp
    > WHERE JSON_EXTRACT(c, "$.id") > 1
    > ORDER BY JSON_EXTRACT(c, "$.name");
+-----------------------------+-----------+------+
| c                           | c->"$.id" | g    |
+-----------------------------+-----------+------+
| {"id": "3", "name": "Barney"} | "3"       |    3 |
| {"id": "4", "name": "Betty"}  | "4"       |    4 |
| {"id": "2", "name": "Wilma"}  | "2"       |    2 |
+-----------------------------+-----------+------+
```

```
3 rows in set (0.00 sec)

mysql> SELECT c, c->"$.id", g
     > FROM jemp
     > WHERE c->"$.id" > 1
     > ORDER BY c->"$.name";
+-----------------------------+-----------+------+
| c                           | c->"$.id" | g    |
+-----------------------------+-----------+------+
| {"id": "3", "name": "Barney"} | "3"       |    3 |
| {"id": "4", "name": "Betty"}  | "4"       |    4 |
| {"id": "2", "name": "Wilma"}  | "2"       |    2 |
+-----------------------------+-----------+------+
3 rows in set (0.00 sec)
```

This functionality is not limited to `SELECT`, as shown here:

```
mysql> ALTER TABLE jemp ADD COLUMN n INT;
Query OK, 0 rows affected (0.68 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> UPDATE jemp SET n=1 WHERE c->"$.id" = "4";
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT c, c->"$.id", g, n
     > FROM jemp
     > WHERE JSON_EXTRACT(c, "$.id") > 1
     > ORDER BY c->"$.name";
+-----------------------------+-----------+------+------+
| c                           | c->"$.id" | g    | n    |
+-----------------------------+-----------+------+------+
| {"id": "3", "name": "Barney"} | "3"       |    3 | NULL |
| {"id": "4", "name": "Betty"}  | "4"       |    4 |    1 |
| {"id": "2", "name": "Wilma"}  | "2"       |    2 | NULL |
+-----------------------------+-----------+------+------+
3 rows in set (0.00 sec)

mysql> DELETE FROM jemp WHERE c->"$.id" = "4";
Query OK, 1 row affected (0.04 sec)

mysql> SELECT c, c->"$.id", g, n
     > FROM jemp
     > WHERE JSON_EXTRACT(c, "$.id") > 1
     > ORDER BY c->"$.name";
+-----------------------------+-----------+------+------+
| c                           | c->"$.id" | g    | n    |
+-----------------------------+-----------+------+------+
| {"id": "3", "name": "Barney"} | "3"       |    3 | NULL |
| {"id": "2", "name": "Wilma"}  | "2"       |    2 | NULL |
+-----------------------------+-----------+------+------+
2 rows in set (0.00 sec)
```

(See Indexing a Generated Column to Provide a JSON Column Index, for the statements used to create and populate the table just shown.)

This also works with JSON array values, as shown here:

```
mysql> CREATE TABLE tj10 (a JSON, b INT);
Query OK, 0 rows affected (0.26 sec)

mysql> INSERT INTO tj10
     > VALUES ("[3,10,5,17,44]", 33), ("[3,10,5,17,[22,44,66]]", 0);
Query OK, 1 row affected (0.04 sec)

mysql> SELECT a->"$[4]" FROM tj10;
+--------------+
| a->"$[4]"    |
+--------------+
| 44           |
| [22, 44, 66] |
```

```
+-------------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM tj10 WHERE a->"$[0]" = 3;
+-----------------------------+------+
| a                           | b    |
+-----------------------------+------+
| [3, 10, 5, 17, 44]          |   33 |
| [3, 10, 5, 17, [22, 44, 66]] |    0 |
+-----------------------------+------+
2 rows in set (0.00 sec)
```

Nested arrays are supported. An expression using `->` evaluates as `NULL` if no matching key is found in the target JSON document, as shown here:

```
mysql> SELECT * FROM tj10 WHERE a->"$[4][1]" IS NOT NULL;
+-----------------------------+------+
| a                           | b    |
+-----------------------------+------+
| [3, 10, 5, 17, [22, 44, 66]] |    0 |
+-----------------------------+------+

mysql> SELECT a->"$[4][1]" FROM tj10;
+-------------+
| a->"$[4][1]" |
+-------------+
| NULL        |
| 44          |
+-------------+
2 rows in set (0.00 sec)
```

This is the same behavior as seen in such cases when using `JSON_EXTRACT()`:

```
mysql> SELECT JSON_EXTRACT(a, "$[4][1]") FROM tj10;
+---------------------------+
| JSON_EXTRACT(a, "$[4][1]") |
+---------------------------+
| NULL                      |
| 44                        |
+---------------------------+
2 rows in set (0.00 sec)
```

- *`column->>path`*

  This is an improved, unquoting extraction operator. Whereas the `->` operator simply extracts a value, the `->>` operator in addition unquotes the extracted result. In other words, given a `JSON` column value *`column`* and a path expression *`path`* (a string literal), the following three expressions return the same value:

  - `JSON_UNQUOTE( JSON_EXTRACT(`*`column, path`*`) )`
  - `JSON_UNQUOTE(`*`column -> path`*`)`
  - *`column->>path`*

  The `->>` operator can be used wherever `JSON_UNQUOTE(JSON_EXTRACT())` would be allowed. This includes (but is not limited to) `SELECT` lists, `WHERE` and `HAVING` clauses, and `ORDER BY` and `GROUP BY` clauses.

  The next few statements demonstrate some `->>` operator equivalences with other expressions in the `mysql` client:

```
mysql> SELECT * FROM jemp WHERE g > 2;
+-----------------------------+------+
| c                           | g    |
+-----------------------------+------+
| {"id": "3", "name": "Barney"} |    3 |
| {"id": "4", "name": "Betty"}  |    4 |
```

```
+------------------------------+------+
2 rows in set (0.01 sec)

mysql> SELECT c->'$.name' AS name
    ->     FROM jemp WHERE g > 2;
+----------+
| name     |
+----------+
| "Barney" |
| "Betty"  |
+----------+
2 rows in set (0.00 sec)

mysql> SELECT JSON_UNQUOTE(c->'$.name') AS name
    ->     FROM jemp WHERE g > 2;
+--------+
| name   |
+--------+
| Barney |
| Betty  |
+--------+
2 rows in set (0.00 sec)

mysql> SELECT c->>'$.name' AS name
    ->     FROM jemp WHERE g > 2;
+--------+
| name   |
+--------+
| Barney |
| Betty  |
+--------+
2 rows in set (0.00 sec)
```

See Indexing a Generated Column to Provide a JSON Column Index, for the SQL statements used to create and populate the `jemp` table in the set of examples just shown.

This operator can also be used with JSON arrays, as shown here:

```
mysql> CREATE TABLE tj10 (a JSON, b INT);
Query OK, 0 rows affected (0.26 sec)

mysql> INSERT INTO tj10 VALUES
    ->     ('[3,10,5,"x",44]', 33),
    ->     ('[3,10,5,17,[22,"y",66]]', 0);
Query OK, 2 rows affected (0.04 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT a->"$[3]", a->"$[4][1]" FROM tj10;
+-----------+--------------+
| a->"$[3]" | a->"$[4][1]" |
+-----------+--------------+
| "x"       | NULL         |
| 17        | "y"          |
+-----------+--------------+
2 rows in set (0.00 sec)

mysql> SELECT a->>"$[3]", a->>"$[4][1]" FROM tj10;
+------------+---------------+
| a->>"$[3]" | a->>"$[4][1]" |
+------------+---------------+
| x          | NULL          |
| 17         | y             |
+------------+---------------+
2 rows in set (0.00 sec)
```

As with `->`, the `->>` operator is always expanded in the output of `EXPLAIN`, as the following example demonstrates:

```
mysql> EXPLAIN SELECT c->>'$.name' AS name
    ->     FROM jemp WHERE g > 2\G
*************************** 1. row ***************************
```

```
           id: 1
  select_type: SIMPLE
        table: jemp
   partitions: NULL
         type: range
possible_keys: i
          key: i
      key_len: 5
          ref: NULL
         rows: 2
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select
json_unquote(json_extract(`jtest`.`jemp`.`c`,'$.name')) AS `name` from
`jtest`.`jemp` where (`jtest`.`jemp`.`g` > 2)
1 row in set (0.00 sec)
```

This is similar to how MySQL expands the `->` operator in the same circumstances.

- `JSON_KEYS(`*`json_doc`*`[, `*`path`*`])`

  Returns the keys from the top-level value of a JSON object as a JSON array, or, if a *`path`* argument is given, the top-level keys from the selected path. Returns `NULL` if any argument is `NULL`, the *`json_doc`* argument is not an object, or *`path`*, if given, does not locate an object. An error occurs if the *`json_doc`* argument is not a valid JSON document or the *`path`* argument is not a valid path expression or contains a `*` or `**` wildcard.

  The result array is empty if the selected object is empty. If the top-level value has nested subobjects, the return value does not include keys from those subobjects.

```
mysql> SELECT JSON_KEYS('{"a": 1, "b": {"c": 30}}');
+---------------------------------------+
| JSON_KEYS('{"a": 1, "b": {"c": 30}}') |
+---------------------------------------+
| ["a", "b"]                            |
+---------------------------------------+
mysql> SELECT JSON_KEYS('{"a": 1, "b": {"c": 30}}', '$.b');
+--------------------------------------------+
| JSON_KEYS('{"a": 1, "b": {"c": 30}}', '$.b') |
+--------------------------------------------+
| ["c"]                                      |
+--------------------------------------------+
```

- `JSON_OVERLAPS(`*`json_doc1, json_doc2`*`)`

  Compares two JSON documents. Returns true (1) if the two document have any key-value pairs or array elements in common. If both arguments are scalars, the function performs a simple equality test. If either argument is `NULL`, the function returns `NULL`.

  This function serves as counterpart to `JSON_CONTAINS()`, which requires all elements of the array searched for to be present in the array searched in. Thus, `JSON_CONTAINS()` performs an `AND` operation on search keys, while `JSON_OVERLAPS()` performs an `OR` operation.

  Queries on JSON columns of `InnoDB` tables using `JSON_OVERLAPS()` in the `WHERE` clause can be optimized using multi-valued indexes. Multi-Valued Indexes, provides detailed information and examples.

  When comparing two arrays, `JSON_OVERLAPS()` returns true if they share one or more array elements in common, and false if they do not:

```
mysql> SELECT JSON_OVERLAPS("[1,3,5,7]", "[2,5,7]");
```

```
+--------------------------------------+
| JSON_OVERLAPS("[1,3,5,7]", "[2,5,7]") |
+--------------------------------------+
|                                    1 |
+--------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT JSON_OVERLAPS("[1,3,5,7]", "[2,6,7]");
+--------------------------------------+
| JSON_OVERLAPS("[1,3,5,7]", "[2,6,7]") |
+--------------------------------------+
|                                    1 |
+--------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT JSON_OVERLAPS("[1,3,5,7]", "[2,6,8]");
+--------------------------------------+
| JSON_OVERLAPS("[1,3,5,7]", "[2,6,8]") |
+--------------------------------------+
|                                    0 |
+--------------------------------------+
1 row in set (0.00 sec)
```

Partial matches are treated as no match, as shown here:

```
mysql> SELECT JSON_OVERLAPS('[[1,2],[3,4],5]', '[1,[2,3],[4,5]]');
+-----------------------------------------------------+
| JSON_OVERLAPS('[[1,2],[3,4],5]', '[1,[2,3],[4,5]]') |
+-----------------------------------------------------+
|                                                   0 |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

When comparing objects, the result is true if they have at least one key-value pair in common.

```
mysql> SELECT JSON_OVERLAPS('{"a":1,"b":10,"d":10}', '{"c":1,"e":10,"f":1,"d":10}');
+-------------------------------------------------------------------------+
| JSON_OVERLAPS('{"a":1,"b":10,"d":10}', '{"c":1,"e":10,"f":1,"d":10}') |
+-------------------------------------------------------------------------+
|                                                                       1 |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT JSON_OVERLAPS('{"a":1,"b":10,"d":10}', '{"a":5,"e":10,"f":1,"d":20}');
+-------------------------------------------------------------------------+
| JSON_OVERLAPS('{"a":1,"b":10,"d":10}', '{"a":5,"e":10,"f":1,"d":20}') |
+-------------------------------------------------------------------------+
|                                                                       0 |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

If two scalars are used as the arguments to the function, `JSON_OVERLAPS()` performs a simple test for equality:

```
mysql> SELECT JSON_OVERLAPS('5', '5');
+-------------------------+
| JSON_OVERLAPS('5', '5') |
+-------------------------+
|                       1 |
+-------------------------+
1 row in set (0.00 sec)

mysql> SELECT JSON_OVERLAPS('5', '6');
+-------------------------+
| JSON_OVERLAPS('5', '6') |
+-------------------------+
|                       0 |
+-------------------------+
```

```
1 row in set (0.00 sec)
```

When comparing a scalar with an array, `JSON_OVERLAPS()` attempts to treat the scalar as an array element. In this example, the second argument `6` is interpreted as `[6]`, as shown here:

```
mysql> SELECT JSON_OVERLAPS('[4,5,6,7]', '6');
+---------------------------------+
| JSON_OVERLAPS('[4,5,6,7]', '6') |
+---------------------------------+
|                               1 |
+---------------------------------+
1 row in set (0.00 sec)
```

The function does not perform type conversions:

```
mysql> SELECT JSON_OVERLAPS('[4,5,"6",7]', '6');
+-----------------------------------+
| JSON_OVERLAPS('[4,5,"6",7]', '6') |
+-----------------------------------+
|                                 0 |
+-----------------------------------+
1 row in set (0.00 sec)

mysql> SELECT JSON_OVERLAPS('[4,5,6,7]', '"6"');
+---------------------------------+
| JSON_OVERLAPS('[4,5,6,7]', '"6"') |
+---------------------------------+
|                                 0 |
+---------------------------------+
1 row in set (0.00 sec)
```

`JSON_OVERLAPS()` was added in MySQL 8.0.17.

- `JSON_SEARCH(`*`json_doc, one_or_all, search_str`*`[, escape_char[, path] ...])`

  Returns the path to the given string within a JSON document. Returns `NULL` if any of the *`json_doc`*, *`search_str`*, or *`path`* arguments are `NULL`; no *`path`* exists within the document; or *`search_str`* is not found. An error occurs if the *`json_doc`* argument is not a valid JSON document, any *`path`* argument is not a valid path expression, *`one_or_all`* is not `'one'` or `'all'`, or *`escape_char`* is not a constant expression.

  The *`one_or_all`* argument affects the search as follows:

  - `'one'`: The search terminates after the first match and returns one path string. It is undefined which match is considered first.
  - `'all'`: The search returns all matching path strings such that no duplicate paths are included. If there are multiple strings, they are autowrapped as an array. The order of the array elements is undefined.

  Within the *`search_str`* search string argument, the `%` and `_` characters work as for the `LIKE` operator: `%` matches any number of characters (including zero characters), and `_` matches exactly one character.

  To specify a literal `%` or `_` character in the search string, precede it by the escape character. The default is `\` if the *`escape_char`* argument is missing or `NULL`. Otherwise, *`escape_char`* must be a constant that is empty or one character.

  For more information about matching and escape character behavior, see the description of `LIKE` in Section 12.8.1, “String Comparison Functions and Operators”. For escape character handling, a difference from the `LIKE` behavior is that the escape character for `JSON_SEARCH()` must evaluate to a constant at compile time, not just at execution time. For example, if `JSON_SEARCH()` is used

in a prepared statement and the *`escape_char`* argument is supplied using a `?` parameter, the parameter value might be constant at execution time, but is not at compile time.

*`search_str`* and *`path`* are always interpreted as utf8mb4 strings, regardless of their actual encoding. This is a known issue which is fixed in MySQL 8.0.24 ( Bug #32449181).

```
mysql> SET @j = '["abc", [{"k": "10"}, "def"], {"x":"abc"}, {"y":"bcd"}]';

mysql> SELECT JSON_SEARCH(@j, 'one', 'abc');
+-------------------------------+
| JSON_SEARCH(@j, 'one', 'abc') |
+-------------------------------+
| "$[0]"                        |
+-------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', 'abc');
+-------------------------------+
| JSON_SEARCH(@j, 'all', 'abc') |
+-------------------------------+
| ["$[0]", "$[2].x"]            |
+-------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', 'ghi');
+-------------------------------+
| JSON_SEARCH(@j, 'all', 'ghi') |
+-------------------------------+
| NULL                          |
+-------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10');
+-------------------------------+
| JSON_SEARCH(@j, 'all', '10') |
+-------------------------------+
| "$[1][0].k"                   |
+-------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10', NULL, '$');
+---------------------------------------+
| JSON_SEARCH(@j, 'all', '10', NULL, '$') |
+---------------------------------------+
| "$[1][0].k"                           |
+---------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10', NULL, '$[*]');
+-----------------------------------------+
| JSON_SEARCH(@j, 'all', '10', NULL, '$[*]') |
+-----------------------------------------+
| "$[1][0].k"                             |
+-----------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10', NULL, '$**.k');
+-------------------------------------------+
| JSON_SEARCH(@j, 'all', '10', NULL, '$**.k') |
+-------------------------------------------+
| "$[1][0].k"                               |
+-------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10', NULL, '$[*][0].k');
+-------------------------------------------------+
| JSON_SEARCH(@j, 'all', '10', NULL, '$[*][0].k') |
+-------------------------------------------------+
| "$[1][0].k"                                     |
+-------------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10', NULL, '$[1]');
+-----------------------------------------+
| JSON_SEARCH(@j, 'all', '10', NULL, '$[1]') |
+-----------------------------------------+
| "$[1][0].k"                             |
+-----------------------------------------+
```

```
mysql> SELECT JSON_SEARCH(@j, 'all', '10', NULL, '$[1][0]');
+-------------------------------------------------+
| JSON_SEARCH(@j, 'all', '10', NULL, '$[1][0]') |
+-------------------------------------------------+
| "$[1][0].k"                                     |
+-------------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', 'abc', NULL, '$[2]');
+-----------------------------------------------+
| JSON_SEARCH(@j, 'all', 'abc', NULL, '$[2]') |
+-----------------------------------------------+
| "$[2].x"                                      |
+-----------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%a%');
+-------------------------------+
| JSON_SEARCH(@j, 'all', '%a%') |
+-------------------------------+
| ["$[0]", "$[2].x"]            |
+-------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%b%');
+-------------------------------+
| JSON_SEARCH(@j, 'all', '%b%') |
+-------------------------------+
| ["$[0]", "$[2].x", "$[3].y"]  |
+-------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%b%', NULL, '$[0]');
+-----------------------------------------------+
| JSON_SEARCH(@j, 'all', '%b%', NULL, '$[0]') |
+-----------------------------------------------+
| "$[0]"                                        |
+-----------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%b%', NULL, '$[2]');
+-----------------------------------------------+
| JSON_SEARCH(@j, 'all', '%b%', NULL, '$[2]') |
+-----------------------------------------------+
| "$[2].x"                                      |
+-----------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%b%', NULL, '$[1]');
+-----------------------------------------------+
| JSON_SEARCH(@j, 'all', '%b%', NULL, '$[1]') |
+-----------------------------------------------+
| NULL                                          |
+-----------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%b%', '', '$[1]');
+-----------------------------------------+
| JSON_SEARCH(@j, 'all', '%b%', '', '$[1]') |
+-----------------------------------------+
| NULL                                    |
+-----------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%b%', '', '$[3]');
+-----------------------------------------+
| JSON_SEARCH(@j, 'all', '%b%', '', '$[3]') |
+-----------------------------------------+
| "$[3].y"                                |
+-----------------------------------------+
```

For more information about the JSON path syntax supported by MySQL, including rules governing the wildcard operators `*` and `**`, see JSON Path Syntax.

- `JSON_VALUE(json_doc, path)`

  Extracts a value from a JSON document at the path given in the specified document, and returns the extracted value, optionally converting it to a desired type. The complete syntax is shown here:

```
JSON_VALUE(json_doc, path [RETURNING type] [on_empty] [on_error])

on_empty:
    {NULL | ERROR | DEFAULT value} ON EMPTY

on_error:
    {NULL | ERROR | DEFAULT value} ON ERROR
```

  *`json_doc`* is a valid JSON document. If this is `NULL`, the function returns `NULL`.

  *`path`* is a JSON path pointing to a location in the document. This must be a string literal value.

  *`type`* is one of the following data types:

  - `FLOAT`
  - `DOUBLE`
  - `DECIMAL`
  - `SIGNED`
  - `UNSIGNED`
  - `DATE`
  - `TIME`
  - `DATETIME`
  - `YEAR` (MySQL 8.0.22 and later)

    `YEAR` values of one or two digits are not supported.
  - `CHAR`
  - `JSON`

  The types just listed are the same as the (non-array) types supported by the `CAST()` function.

  If not specified by a `RETURNING` clause, the `JSON_VALUE()` function's return type is `VARCHAR(512)`. When no character set is specified for the return type, `JSON_VALUE()` uses `utf8mb4` with the binary collation, which is case-sensitive; if `utf8mb4` is specified as the character

set for the result, the server uses the default collation for this character set, which is not case-sensitive.

When the data at the specified path consists of or resolves to a JSON null literal, the function returns SQL `NULL`.

*`on_empty`*, if specified, determines how `JSON_VALUE()` behaves when no data is found at the path given; this clause takes one of the following values:

- `NULL ON EMPTY`: The function returns `NULL`; this is the default `ON EMPTY` behavior.
- `DEFAULT value ON EMPTY`: the provided *`value`* is returned. The value's type must match that of the return type.
- `ERROR ON EMPTY`: The function throws an error.

If used, *`on_error`* takes one of the following values with the corresponding outcome when an error occurs, as listed here:

- `NULL ON ERROR`: `JSON_VALUE()` returns `NULL`; this is the default behavior if no `ON ERROR` clause is used.
- `DEFAULT value ON ERROR`: This is the value returned; its value must match that of the return type.
- `ERROR ON ERROR`: An error is thrown.

`ON EMPTY`, if used, must precede any `ON ERROR` clause. Specifying them in the wrong order results in a syntax error.

**Error handling.** In general, errors are handled by `JSON_VALUE()` as follows:

- All JSON input (document and path) is checked for validity. If any of it is not valid, an SQL error is thrown without triggering the `ON ERROR` clause.
- `ON ERROR` is triggered whenever any of the following events occur:
  - Attempting to extract an object or an array, such as that resulting from a path that resolves to multiple locations within the JSON document
  - Conversion errors, such as attempting to convert `'asdf'` to an `UNSIGNED` value
  - Truncation of values
- A conversion error always triggers a warning even if `NULL ON ERROR` or `DEFAULT ... ON ERROR` is specified.
- The `ON EMPTY` clause is triggered when the source JSON document (*`expr`*) contains no data at the specified location (*`path`*).

`JSON_VALUE()` was introduced in MySQL 8.0.21.

**Examples.** Two simple examples are shown here:

```
mysql> SELECT JSON_VALUE('{"fname": "Joe", "lname": "Palmer"}', '$.fname');
+-----------------------------------------------------------+
| JSON_VALUE('{"fname": "Joe", "lname": "Palmer"}', '$.fname') |
+-----------------------------------------------------------+
| Joe                                                       |
+-----------------------------------------------------------+

mysql> SELECT JSON_VALUE('{"item": "shoes", "price": "49.95"}', '$.price'
    -> RETURNING DECIMAL(4,2)) AS price;
```

```
+-------+
| price |
+-------+
| 49.95 |
+-------+
```

The statement `SELECT JSON_VALUE(json_doc, path RETURNING type)` is equivalent to the following statement:

```
SELECT CAST(
    JSON_UNQUOTE( JSON_EXTRACT(json_doc, path) )
    AS type
);
```

`JSON_VALUE()` simplifies creating indexes on JSON columns by making it unnecessary in many cases to create a generated column and then an index on the generated column. You can do this when creating a table `t1` that has a `JSON` column by creating an index on an expression that uses `JSON_VALUE()` operating on that column (with a path that matches a value in that column), as shown here:

```
CREATE TABLE t1(
    j JSON,
    INDEX i1 ( (JSON_VALUE(j, '$.id' RETURNING UNSIGNED)) )
);
```

The following `EXPLAIN` output shows that a query against `t1` employing the index expression in the `WHERE` clause uses the index thus created:

```
mysql> EXPLAIN SELECT * FROM t1
    ->     WHERE JSON_VALUE(j, '$.id' RETURNING UNSIGNED) = 123\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: ref
possible_keys: i1
          key: i1
      key_len: 9
          ref: const
         rows: 1
     filtered: 100.00
        Extra: NULL
```

This achieves much the same effect as creating a table `t2` with an index on a generated column (see Indexing a Generated Column to Provide a JSON Column Index), like this one:

```
CREATE TABLE t2 (
    j JSON,
    g INT GENERATED ALWAYS AS (j->"$.id"),
    INDEX i1 (g)
);
```

The `EXPLAIN` output for a query against this table, referencing the generated column, shows that the index is used in the same way as for the previous query against table `t1`:

```
mysql> EXPLAIN SELECT * FROM t2 WHERE g  = 123\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t2
   partitions: NULL
         type: ref
possible_keys: i1
          key: i1
      key_len: 5
          ref: const
         rows: 1
     filtered: 100.00
```

```
        Extra: NULL
```

For information about using indexes on generated columns for indirect indexing of `JSON` columns, see Indexing a Generated Column to Provide a JSON Column Index.

- *`value`* `MEMBER OF(`*`json_array`*`)`

  Returns true (1) if *`value`* is an element of *`json_array`*, otherwise returns false (0). *`value`* must be a scalar or a JSON document; if it is a scalar, the operator attempts to treat it as an element of a JSON array. If *`value`* or *`json_array`* is *`NULL`*, the function returns *`NULL`*.

  Queries using `MEMBER OF()` on JSON columns of `InnoDB` tables in the `WHERE` clause can be optimized using multi-valued indexes. See Multi-Valued Indexes, for detailed information and examples.

  Simple scalars are treated as array values, as shown here:

```
mysql> SELECT 17 MEMBER OF('[23, "abc", 17, "ab", 10]');
+-------------------------------------------+
| 17 MEMBER OF('[23, "abc", 17, "ab", 10]') |
+-------------------------------------------+
|                                         1 |
+-------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT 'ab' MEMBER OF('[23, "abc", 17, "ab", 10]');
+---------------------------------------------+
| 'ab' MEMBER OF('[23, "abc", 17, "ab", 10]') |
+---------------------------------------------+
|                                           1 |
+---------------------------------------------+
1 row in set (0.00 sec)
```

  Partial matches of array element values do not match:

```
mysql> SELECT 7 MEMBER OF('[23, "abc", 17, "ab", 10]');
+------------------------------------------+
| 7 MEMBER OF('[23, "abc", 17, "ab", 10]') |
+------------------------------------------+
|                                        0 |
+------------------------------------------+
1 row in set (0.00 sec)
```

```
mysql> SELECT 'a' MEMBER OF('[23, "abc", 17, "ab", 10]');
+--------------------------------------------+
| 'a' MEMBER OF('[23, "abc", 17, "ab", 10]') |
+--------------------------------------------+
|                                          0 |
+--------------------------------------------+
1 row in set (0.00 sec)
```

  Conversions to and from string types are not performed:

```
mysql> SELECT
    -> 17 MEMBER OF('[23, "abc", "17", "ab", 10]'),
    -> "17" MEMBER OF('[23, "abc", 17, "ab", 10]')\G
*************************** 1. row ***************************
17 MEMBER OF('[23, "abc", "17", "ab", 10]'): 0
"17" MEMBER OF('[23, "abc", 17, "ab", 10]'): 0
1 row in set (0.00 sec)
```

  To use this operator with a value which itself an array, it is necessary to cast it explicitly as a JSON array. You can do this with `CAST(... AS JSON)`:

```
mysql> SELECT CAST('[4,5]' AS JSON) MEMBER OF('[[3,4],[4,5]]');
+------------------------------------------------+
| CAST('[4,5]' AS JSON) MEMBER OF('[[3,4],[4,5]]') |
+------------------------------------------------+
```

```
|                                                 1 |
+---------------------------------------------------+
1 row in set (0.00 sec)
```

It is also possible to perform the necessary cast using the `JSON_ARRAY()` function, like this:

```
mysql> SELECT JSON_ARRAY(4,5) MEMBER OF('[[3,4],[4,5]]');
+-------------------------------------------+
| JSON_ARRAY(4,5) MEMBER OF('[[3,4],[4,5]]') |
+-------------------------------------------+
|                                         1 |
+-------------------------------------------+
1 row in set (0.00 sec)
```

Any JSON objects used as values to be tested or which appear in the target array must be coerced to the correct type using `CAST(... AS JSON)` or `JSON_OBJECT()`. In addition, a target array containing JSON objects must itself be cast using `JSON_ARRAY`. This is demonstrated in the following sequence of statements:

```
mysql> SET @a = CAST('{"a":1}' AS JSON);
Query OK, 0 rows affected (0.00 sec)

mysql> SET @b = JSON_OBJECT("b", 2);
Query OK, 0 rows affected (0.00 sec)

mysql> SET @c = JSON_ARRAY(17, @b, "abc", @a, 23);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @a MEMBER OF(@c), @b MEMBER OF(@c);
+------------------+------------------+
| @a MEMBER OF(@c) | @b MEMBER OF(@c) |
+------------------+------------------+
|                1 |                1 |
+------------------+------------------+
1 row in set (0.00 sec)
```

The `MEMBER OF()` operator was added in MySQL 8.0.17.

## 12.18.4 Functions That Modify JSON Values

The functions in this section modify JSON values and return the result.

- `JSON_ARRAY_APPEND(`*`json_doc`*`, `*`path`*`, `*`val`*`[, `*`path`*`, `*`val`*`] ...)`

  Appends values to the end of the indicated arrays within a JSON document and returns the result. Returns `NULL` if any argument is `NULL`. An error occurs if the *`json_doc`* argument is not a valid JSON document or any *`path`* argument is not a valid path expression or contains a `*` or `**` wildcard.

  The path-value pairs are evaluated left to right. The document produced by evaluating one pair becomes the new value against which the next pair is evaluated.

  If a path selects a scalar or object value, that value is autowrapped within an array and the new value is added to that array. Pairs for which the path does not identify any value in the JSON document are ignored.

```
mysql> SET @j = '["a", ["b", "c"], "d"]';
mysql> SELECT JSON_ARRAY_APPEND(@j, '$[1]', 1);
+---------------------------------+
| JSON_ARRAY_APPEND(@j, '$[1]', 1) |
+---------------------------------+
| ["a", ["b", "c", 1], "d"]       |
+---------------------------------+
mysql> SELECT JSON_ARRAY_APPEND(@j, '$[0]', 2);
+---------------------------------+
| JSON_ARRAY_APPEND(@j, '$[0]', 2) |
+---------------------------------+
| [["a", 2], ["b", "c"], "d"]     |
+---------------------------------+
```

```
mysql> SELECT JSON_ARRAY_APPEND(@j, '$[1][0]', 3);
+-----------------------------------+
| JSON_ARRAY_APPEND(@j, '$[1][0]', 3) |
+-----------------------------------+
| ["a", [["b", 3], "c"], "d"]         |
+-----------------------------------+

mysql> SET @j = '{"a": 1, "b": [2, 3], "c": 4}';
mysql> SELECT JSON_ARRAY_APPEND(@j, '$.b', 'x');
+-----------------------------------+
| JSON_ARRAY_APPEND(@j, '$.b', 'x')  |
+-----------------------------------+
| {"a": 1, "b": [2, 3, "x"], "c": 4} |
+-----------------------------------+
mysql> SELECT JSON_ARRAY_APPEND(@j, '$.c', 'y');
+-----------------------------------+
| JSON_ARRAY_APPEND(@j, '$.c', 'y')    |
+-----------------------------------+
| {"a": 1, "b": [2, 3], "c": [4, "y"]} |
+-----------------------------------+

mysql> SET @j = '{"a": 1}';
mysql> SELECT JSON_ARRAY_APPEND(@j, '$', 'z');
+--------------------------------+
| JSON_ARRAY_APPEND(@j, '$', 'z') |
+--------------------------------+
| [{"a": 1}, "z"]                 |
+--------------------------------+
```

In MySQL 5.7, this function was named `JSON_APPEND()`. That name is no longer supported in MySQL 8.0.

- `JSON_ARRAY_INSERT(`*`json_doc`*`, `*`path`*`, `*`val`*`[, `*`path`*`, `*`val`*`] ...)`

  Updates a JSON document, inserting into an array within the document and returning the modified document. Returns `NULL` if any argument is `NULL`. An error occurs if the *`json_doc`* argument is not a valid JSON document or any *`path`* argument is not a valid path expression or contains a `*` or `**` wildcard or does not end with an array element identifier.

  The path-value pairs are evaluated left to right. The document produced by evaluating one pair becomes the new value against which the next pair is evaluated.

  Pairs for which the path does not identify any array in the JSON document are ignored. If a path identifies an array element, the corresponding value is inserted at that element position, shifting any following values to the right. If a path identifies an array position past the end of an array, the value is inserted at the end of the array.

```
mysql> SET @j = '["a", {"b": [1, 2]}, [3, 4]]';
mysql> SELECT JSON_ARRAY_INSERT(@j, '$[1]', 'x');
+-----------------------------------+
| JSON_ARRAY_INSERT(@j, '$[1]', 'x') |
+-----------------------------------+
| ["a", "x", {"b": [1, 2]}, [3, 4]]  |
+-----------------------------------+
mysql> SELECT JSON_ARRAY_INSERT(@j, '$[100]', 'x');
+-------------------------------------+
| JSON_ARRAY_INSERT(@j, '$[100]', 'x') |
+-------------------------------------+
| ["a", {"b": [1, 2]}, [3, 4], "x"]    |
+-------------------------------------+
mysql> SELECT JSON_ARRAY_INSERT(@j, '$[1].b[0]', 'x');
+----------------------------------------+
| JSON_ARRAY_INSERT(@j, '$[1].b[0]', 'x') |
+----------------------------------------+
| ["a", {"b": ["x", 1, 2]}, [3, 4]]       |
+----------------------------------------+
mysql> SELECT JSON_ARRAY_INSERT(@j, '$[2][1]', 'y');
+-------------------------------------+
| JSON_ARRAY_INSERT(@j, '$[2][1]', 'y') |
```

```
+-------------------------------------+
| ["a", {"b": [1, 2]}, [3, "y", 4]]     |
+-------------------------------------+
mysql> SELECT JSON_ARRAY_INSERT(@j, '$[0]', 'x', '$[2][1]', 'y');
+-------------------------------------------------+
| JSON_ARRAY_INSERT(@j, '$[0]', 'x', '$[2][1]', 'y') |
+-------------------------------------------------+
| ["x", "a", {"b": [1, 2]}, [3, 4]]                 |
+-------------------------------------------------+
```

Earlier modifications affect the positions of the following elements in the array, so subsequent paths in the same `JSON_ARRAY_INSERT()` call should take this into account. In the final example, the second path inserts nothing because the path no longer matches anything after the first insert.

- `JSON_INSERT(json_doc, path, val[, path, val] ...)`

  Inserts data into a JSON document and returns the result. Returns `NULL` if any argument is `NULL`. An error occurs if the *`json_doc`* argument is not a valid JSON document or any *`path`* argument is not a valid path expression or contains a `*` or `**` wildcard.

  The path-value pairs are evaluated left to right. The document produced by evaluating one pair becomes the new value against which the next pair is evaluated.

  A path-value pair for an existing path in the document is ignored and does not overwrite the existing document value. A path-value pair for a nonexisting path in the document adds the value to the document if the path identifies one of these types of values:

  - A member not present in an existing object. The member is added to the object and associated with the new value.
  - A position past the end of an existing array. The array is extended with the new value. If the existing value is not an array, it is autowrapped as an array, then extended with the new value.

  Otherwise, a path-value pair for a nonexisting path in the document is ignored and has no effect.

  For a comparison of `JSON_INSERT()`, `JSON_REPLACE()`, and `JSON_SET()`, see the discussion of `JSON_SET()`.

```
mysql> SET @j = '{ "a": 1, "b": [2, 3]}';
mysql> SELECT JSON_INSERT(@j, '$.a', 10, '$.c', '[true, false]');
+----------------------------------------------------+
| JSON_INSERT(@j, '$.a', 10, '$.c', '[true, false]') |
+----------------------------------------------------+
| {"a": 1, "b": [2, 3], "c": "[true, false]"}        |
+----------------------------------------------------+
```

The third and final value listed in the result is a quoted string and not an array like the second one (which is not quoted in the output); no casting of values to the JSON type is performed. To insert the array as an array, you must perform such casts explicitly, as shown here:

```
mysql> SELECT JSON_INSERT(@j, '$.a', 10, '$.c', CAST('[true, false]' AS JSON));
+-----------------------------------------------------------------+
| JSON_INSERT(@j, '$.a', 10, '$.c', CAST('[true, false]' AS JSON)) |
+-----------------------------------------------------------------+
| {"a": 1, "b": [2, 3], "c": [true, false]}                       |
+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

- `JSON_MERGE(json_doc, json_doc[, json_doc] ...)`

  Merges two or more JSON documents. Synonym for `JSON_MERGE_PRESERVE()`; deprecated in MySQL 8.0.3 and subject to removal in a future release.

```
mysql> SELECT JSON_MERGE('[1, 2]', '[true, false]');
+---------------------------------------+
| JSON_MERGE('[1, 2]', '[true, false]') |
```

```
+-------------------------------------+
| [1, 2, true, false]                 |
+-------------------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
   Code: 1287
Message: 'JSON_MERGE' is deprecated and will be removed in a future release. \
 Please use JSON_MERGE_PRESERVE/JSON_MERGE_PATCH instead
1 row in set (0.00 sec)
```

For additional examples, see the entry for `JSON_MERGE_PRESERVE()`.

- `JSON_MERGE_PATCH(`*`json_doc, json_doc`*`[, `*`json_doc`*`] ...)`

  Performs an RFC 7396 compliant merge of two or more JSON documents and returns the merged result, without preserving members having duplicate keys. Raises an error if at least one of the documents passed as arguments to this function is not valid.

  

  **Note**

  For an explanation and example of the differences between this function and `JSON_MERGE_PRESERVE()`, see JSON_MERGE_PATCH() compared with JSON_MERGE_PRESERVE().

  `JSON_MERGE_PATCH()` performs a merge as follows:

  1. If the first argument is not an object, the result of the merge is the same as if an empty object had been merged with the second argument.
  2. If the second argument is not an object, the result of the merge is the second argument.
  3. If both arguments are objects, the result of the merge is an object with the following members:
     - All members of the first object which do not have a corresponding member with the same key in the second object.
     - All members of the second object which do not have a corresponding key in the first object, and whose value is not the JSON `null` literal.
     - All members with a key that exists in both the first and the second object, and whose value in the second object is not the JSON `null` literal. The values of these members are the results of recursively merging the value in the first object with the value in the second object.

  For additional information, see Normalization, Merging, and Autowrapping of JSON Values.

```
mysql> SELECT JSON_MERGE_PATCH('[1, 2]', '[true, false]');
+-------------------------------------------+
| JSON_MERGE_PATCH('[1, 2]', '[true, false]') |
+-------------------------------------------+
| [true, false]                             |
+-------------------------------------------+

mysql> SELECT JSON_MERGE_PATCH('{"name": "x"}', '{"id": 47}');
+-------------------------------------------------+
| JSON_MERGE_PATCH('{"name": "x"}', '{"id": 47}') |
+-------------------------------------------------+
| {"id": 47, "name": "x"}                         |
+-------------------------------------------------+

mysql> SELECT JSON_MERGE_PATCH('1', 'true');
+------------------------------+
| JSON_MERGE_PATCH('1', 'true') |
+------------------------------+
```

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

mysql> SELECT JSON_MERGE_PATCH('[1, 2]', '{"id": 47}');
+----------------------------------------+
| JSON_MERGE_PATCH('[1, 2]', '{"id": 47}') |
+----------------------------------------+
| {"id": 47}                             |
+----------------------------------------+

mysql> SELECT JSON_MERGE_PATCH('{ "a": 1, "b":2 }',
    >     '{ "a": 3, "c":4 }');
+-------------------------------------------------------+
| JSON_MERGE_PATCH('{ "a": 1, "b":2 }','{ "a": 3, "c":4 }') |
+-------------------------------------------------------+
| {"a": 3, "b": 2, "c": 4}                               |
+-------------------------------------------------------+

mysql> SELECT JSON_MERGE_PATCH('{ "a": 1, "b":2 }','{ "a": 3, "c":4 }',
    >     '{ "a": 5, "d":6 }');
+-------------------------------------------------------------------------------+
| JSON_MERGE_PATCH('{ "a": 1, "b":2 }','{ "a": 3, "c":4 }','{ "a": 5, "d":6 }') |
+-------------------------------------------------------------------------------+
| {"a": 5, "b": 2, "c": 4, "d": 6}                                              |
+-------------------------------------------------------------------------------+
```

You can use this function to remove a member by specifying `null` as the value of the same member in the second argument, as shown here:

```
mysql> SELECT JSON_MERGE_PATCH('{"a":1, "b":2}', '{"b":null}');
+-----------------------------------------------+
| JSON_MERGE_PATCH('{"a":1, "b":2}', '{"b":null}') |
+-----------------------------------------------+
| {"a": 1}                                      |
+-----------------------------------------------+
```

This example shows that the function operates in a recursive fashion; that is, values of members are not limited to scalars, but rather can themselves be JSON documents:

```
mysql> SELECT JSON_MERGE_PATCH('{"a":{"x":1}}', '{"a":{"y":2}}');
+-----------------------------------------------+
| JSON_MERGE_PATCH('{"a":{"x":1}}', '{"a":{"y":2}}') |
+-----------------------------------------------+
| {"a": {"x": 1, "y": 2}}                       |
+-----------------------------------------------+
```

`JSON_MERGE_PATCH()` is supported in MySQL 8.0.3 and later.

**JSON_MERGE_PATCH() compared with JSON_MERGE_PRESERVE().** The behavior of `JSON_MERGE_PATCH()` is the same as that of `JSON_MERGE_PRESERVE()`, with the following two exceptions:

- `JSON_MERGE_PATCH()` removes any member in the first object with a matching key in the second object, provided that the value associated with the key in the second object is not JSON `null`.
- If the second object has a member with a key matching a member in the first object, `JSON_MERGE_PATCH()` *replaces* the value in the first object with the value in the second object, whereas `JSON_MERGE_PRESERVE()` *appends* the second value to the first value.

This example compares the results of merging the same 3 JSON objects, each having a matching key `"a"`, with each of these two functions:

```
mysql> SET @x = '{ "a": 1, "b": 2 }',
    >     @y = '{ "a": 3, "c": 4 }',
    >     @z = '{ "a": 5, "d": 6 }';

mysql> SELECT  JSON_MERGE_PATCH(@x, @y, @z)    AS Patch,
    ->         JSON_MERGE_PRESERVE(@x, @y, @z) AS Preserve\G
```

```
*************************** 1. row ***************************
   Patch: {"a": 5, "b": 2, "c": 4, "d": 6}
Preserve: {"a": [1, 3, 5], "b": 2, "c": 4, "d": 6}
```

- `JSON_MERGE_PRESERVE(`*`json_doc`*`, `*`json_doc`*`[, `*`json_doc`*`] ...)`

  Merges two or more JSON documents and returns the merged result. Returns `NULL` if any argument is `NULL`. An error occurs if any argument is not a valid JSON document.

  Merging takes place according to the following rules. For additional information, see Normalization, Merging, and Autowrapping of JSON Values.

  - Adjacent arrays are merged to a single array.
  - Adjacent objects are merged to a single object.
  - A scalar value is autowrapped as an array and merged as an array.
  - An adjacent array and object are merged by autowrapping the object as an array and merging the two arrays.

```
mysql> SELECT JSON_MERGE_PRESERVE('[1, 2]', '[true, false]');
+-----------------------------------------------+
| JSON_MERGE_PRESERVE('[1, 2]', '[true, false]') |
+-----------------------------------------------+
| [1, 2, true, false]                           |
+-----------------------------------------------+

mysql> SELECT JSON_MERGE_PRESERVE('{"name": "x"}', '{"id": 47}');
+-------------------------------------------------+
| JSON_MERGE_PRESERVE('{"name": "x"}', '{"id": 47}') |
+-------------------------------------------------+
| {"id": 47, "name": "x"}                         |
+-------------------------------------------------+

mysql> SELECT JSON_MERGE_PRESERVE('1', 'true');
+---------------------------------+
| JSON_MERGE_PRESERVE('1', 'true') |
+---------------------------------+
| [1, true]                       |
+---------------------------------+

mysql> SELECT JSON_MERGE_PRESERVE('[1, 2]', '{"id": 47}');
+-------------------------------------------+
| JSON_MERGE_PRESERVE('[1, 2]', '{"id": 47}') |
+-------------------------------------------+
| [1, 2, {"id": 47}]                        |
+-------------------------------------------+

mysql> SELECT JSON_MERGE_PRESERVE('{ "a": 1, "b": 2 }',
    >    '{ "a": 3, "c": 4 }');
+-------------------------------------------------------------+
| JSON_MERGE_PRESERVE('{ "a": 1, "b": 2 }','{ "a": 3, "c":4 }') |
+-------------------------------------------------------------+
| {"a": [1, 3], "b": 2, "c": 4}                               |
+-------------------------------------------------------------+

mysql> SELECT JSON_MERGE_PRESERVE('{ "a": 1, "b": 2 }','{ "a": 3, "c": 4 }',
    >    '{ "a": 5, "d": 6 }');
+-----------------------------------------------------------------------------------+
| JSON_MERGE_PRESERVE('{ "a": 1, "b": 2 }','{ "a": 3, "c": 4 }','{ "a": 5, "d": 6 }') |
+-----------------------------------------------------------------------------------+
| {"a": [1, 3, 5], "b": 2, "c": 4, "d": 6}                                          |
```

```
+--------------------------------------------------------------------------------+
```

This function was added in MySQL 8.0.3 as a synonym for `JSON_MERGE()`. The `JSON_MERGE()` function is now deprecated, and is subject to removal in a future release of MySQL.

This function is similar to but differs from `JSON_MERGE_PATCH()` in significant respects; see JSON_MERGE_PATCH() compared with JSON_MERGE_PRESERVE(), for more information.

- `JSON_REMOVE(json_doc, path[, path] ...)`

  Removes data from a JSON document and returns the result. Returns `NULL` if any argument is `NULL`. An error occurs if the *`json_doc`* argument is not a valid JSON document or any *`path`* argument is not a valid path expression or is `$` or contains a `*` or `**` wildcard.

  The *`path`* arguments are evaluated left to right. The document produced by evaluating one path becomes the new value against which the next path is evaluated.

  It is not an error if the element to be removed does not exist in the document; in that case, the path does not affect the document.

```
mysql> SET @j = '["a", ["b", "c"], "d"]';
mysql> SELECT JSON_REMOVE(@j, '$[1]');
+-------------------------+
| JSON_REMOVE(@j, '$[1]') |
+-------------------------+
| ["a", "d"]              |
+-------------------------+
```

- `JSON_REPLACE(json_doc, path, val[, path, val] ...)`

  Replaces existing values in a JSON document and returns the result. Returns `NULL` if any argument is `NULL`. An error occurs if the *`json_doc`* argument is not a valid JSON document or any *`path`* argument is not a valid path expression or contains a `*` or `**` wildcard.

  The path-value pairs are evaluated left to right. The document produced by evaluating one pair becomes the new value against which the next pair is evaluated.

  A path-value pair for an existing path in the document overwrites the existing document value with the new value. A path-value pair for a nonexisting path in the document is ignored and has no effect.

  In MySQL 8.0.4, the optimizer can perform a partial, in-place update of a `JSON` column instead of removing the old document and writing the new document in its entirety to the column. This optimization can be performed for an update statement that uses the `JSON_REPLACE()` function and meets the conditions outlined in Partial Updates of JSON Values.

  For a comparison of `JSON_INSERT()`, `JSON_REPLACE()`, and `JSON_SET()`, see the discussion of `JSON_SET()`.

```
mysql> SET @j = '{ "a": 1, "b": [2, 3]}';
mysql> SELECT JSON_REPLACE(@j, '$.a', 10, '$.c', '[true, false]');
+-----------------------------------------------------+
| JSON_REPLACE(@j, '$.a', 10, '$.c', '[true, false]') |
+-----------------------------------------------------+
| {"a": 10, "b": [2, 3]}                              |
+-----------------------------------------------------+
```

- `JSON_SET(json_doc, path, val[, path, val] ...)`

  Inserts or updates data in a JSON document and returns the result. Returns `NULL` if *`json_doc`* or *`path`* is `NULL`, or if *`path`*, when given, does not locate an object. Otherwise, an error occurs if

the *`json_doc`* argument is not a valid JSON document or any *`path`* argument is not a valid path expression or contains a `*` or `**` wildcard.

The path-value pairs are evaluated left to right. The document produced by evaluating one pair becomes the new value against which the next pair is evaluated.

A path-value pair for an existing path in the document overwrites the existing document value with the new value. A path-value pair for a nonexisting path in the document adds the value to the document if the path identifies one of these types of values:

- A member not present in an existing object. The member is added to the object and associated with the new value.
- A position past the end of an existing array. The array is extended with the new value. If the existing value is not an array, it is autowrapped as an array, then extended with the new value.

Otherwise, a path-value pair for a nonexisting path in the document is ignored and has no effect.

In MySQL 8.0.4, the optimizer can perform a partial, in-place update of a `JSON` column instead of removing the old document and writing the new document in its entirety to the column. This optimization can be performed for an update statement that uses the `JSON_SET()` function and meets the conditions outlined in Partial Updates of JSON Values.

The `JSON_SET()`, `JSON_INSERT()`, and `JSON_REPLACE()` functions are related:

- `JSON_SET()` replaces existing values and adds nonexisting values.
- `JSON_INSERT()` inserts values without replacing existing values.
- `JSON_REPLACE()` replaces *only* existing values.

The following examples illustrate these differences, using one path that does exist in the document (`$.a`) and another that does not exist (`$.c`):

```
mysql> SET @j = '{ "a": 1, "b": [2, 3]}';
mysql> SELECT JSON_SET(@j, '$.a', 10, '$.c', '[true, false]');
+-------------------------------------------------+
| JSON_SET(@j, '$.a', 10, '$.c', '[true, false]') |
+-------------------------------------------------+
| {"a": 10, "b": [2, 3], "c": "[true, false]"}    |
+-------------------------------------------------+
mysql> SELECT JSON_INSERT(@j, '$.a', 10, '$.c', '[true, false]');
+----------------------------------------------------+
| JSON_INSERT(@j, '$.a', 10, '$.c', '[true, false]') |
+----------------------------------------------------+
| {"a": 1, "b": [2, 3], "c": "[true, false]"}        |
+----------------------------------------------------+
mysql> SELECT JSON_REPLACE(@j, '$.a', 10, '$.c', '[true, false]');
+-----------------------------------------------------+
| JSON_REPLACE(@j, '$.a', 10, '$.c', '[true, false]') |
+-----------------------------------------------------+
| {"a": 10, "b": [2, 3]}                              |
+-----------------------------------------------------+
```

- `JSON_UNQUOTE(`*`json_val`*`)`

  Unquotes JSON value and returns the result as a `utf8mb4` string. Returns `NULL` if the argument is `NULL`. An error occurs if the value starts and ends with double quotes but is not a valid JSON string literal.

  Within a string, certain sequences have special meaning unless the `NO_BACKSLASH_ESCAPES` SQL mode is enabled. Each of these sequences begins with a backslash (`\`), known as the *escape character*. MySQL recognizes the escape sequences shown in Table 12.23, "JSON_UNQUOTE() Special Character Escape Sequences". For all other escape sequences, backslash is ignored. That

is, the escaped character is interpreted as if it was not escaped. For example, `\x` is just `x`. These sequences are case-sensitive. For example, `\b` is interpreted as a backspace, but `\B` is interpreted as `B`.

**Table 12.23 JSON_UNQUOTE() Special Character Escape Sequences**

| Escape Sequence | Character Represented by Sequence |
|---|---|
| `\"` | A double quote (") character |
| `\b` | A backspace character |
| `\f` | A formfeed character |
| `\n` | A newline (linefeed) character |
| `\r` | A carriage return character |
| `\t` | A tab character |
| `\\` | A backslash (\) character |
| `\u`*XXXX* | UTF-8 bytes for Unicode value *XXXX* |

Two simple examples of the use of this function are shown here:

```
mysql> SET @j = '"abc"';
mysql> SELECT @j, JSON_UNQUOTE(@j);
+-------+------------------+
| @j    | JSON_UNQUOTE(@j) |
+-------+------------------+
| "abc" | abc              |
+-------+------------------+
mysql> SET @j = '[1, 2, 3]';
mysql> SELECT @j, JSON_UNQUOTE(@j);
+-----------+------------------+
| @j        | JSON_UNQUOTE(@j) |
+-----------+------------------+
| [1, 2, 3] | [1, 2, 3]        |
+-----------+------------------+
```

The following set of examples shows how `JSON_UNQUOTE` handles escapes with `NO_BACKSLASH_ESCAPES` disabled and enabled:

```
mysql> SELECT @@sql_mode;
+------------+
| @@sql_mode |
+------------+
|            |
+------------+

mysql> SELECT JSON_UNQUOTE('"\\t\\u0032"');
+------------------------------+
| JSON_UNQUOTE('"\\t\\u0032"') |
+------------------------------+
|       2                          |
+------------------------------+

mysql> SET @@sql_mode = 'NO_BACKSLASH_ESCAPES';
mysql> SELECT JSON_UNQUOTE('"\\t\\u0032"');
+------------------------------+
| JSON_UNQUOTE('"\\t\\u0032"') |
+------------------------------+
| \t\u0032                     |
+------------------------------+

mysql> SELECT JSON_UNQUOTE('"\t\u0032"');
+---------------------------+
| JSON_UNQUOTE('"\t\u0032"') |
+---------------------------+
|       2                       |
+---------------------------+
```

## 12.18.5 Functions That Return JSON Value Attributes

The functions in this section return attributes of JSON values.

- `JSON_DEPTH(json_doc)`

  Returns the maximum depth of a JSON document. Returns `NULL` if the argument is `NULL`. An error occurs if the argument is not a valid JSON document.

  An empty array, empty object, or scalar value has depth 1. A nonempty array containing only elements of depth 1 or nonempty object containing only member values of depth 1 has depth 2. Otherwise, a JSON document has depth greater than 2.

```
mysql> SELECT JSON_DEPTH('{}'), JSON_DEPTH('[]'), JSON_DEPTH('true');
+-----------------+-----------------+-------------------+
| JSON_DEPTH('{}') | JSON_DEPTH('[]') | JSON_DEPTH('true') |
+-----------------+-----------------+-------------------+
|               1 |               1 |                 1 |
+-----------------+-----------------+-------------------+
mysql> SELECT JSON_DEPTH('[10, 20]'), JSON_DEPTH('[[], {}]');
+-----------------------+-----------------------+
| JSON_DEPTH('[10, 20]') | JSON_DEPTH('[[], {}]') |
+-----------------------+-----------------------+
|                     2 |                     2 |
+-----------------------+-----------------------+
mysql> SELECT JSON_DEPTH('[10, {"a": 20}]');
+-------------------------------+
| JSON_DEPTH('[10, {"a": 20}]') |
+-------------------------------+
|                             3 |
+-------------------------------+
```

- `JSON_LENGTH(json_doc[, path])`

  Returns the length of a JSON document, or, if a *`path`* argument is given, the length of the value within the document identified by the path. Returns `NULL` if any argument is `NULL` or the *`path`* argument does not identify a value in the document. An error occurs if the *`json_doc`* argument is not a valid JSON document or the *`path`* argument is not a valid path expression. Prior to MySQL 8.0.26, an error is also raised if the path expression contains a `*` or `**` wildcard.

  The length of a document is determined as follows:

  - The length of a scalar is 1.
  - The length of an array is the number of array elements.
  - The length of an object is the number of object members.
  - The length does not count the length of nested arrays or objects.

```
mysql> SELECT JSON_LENGTH('[1, 2, {"a": 3}]');
+---------------------------------+
| JSON_LENGTH('[1, 2, {"a": 3}]') |
+---------------------------------+
|                               3 |
+---------------------------------+
mysql> SELECT JSON_LENGTH('{"a": 1, "b": {"c": 30}}');
+---------------------------------------+
| JSON_LENGTH('{"a": 1, "b": {"c": 30}}') |
+---------------------------------------+
|                                     2 |
+---------------------------------------+
mysql> SELECT JSON_LENGTH('{"a": 1, "b": {"c": 30}}', '$.b');
+------------------------------------------------+
| JSON_LENGTH('{"a": 1, "b": {"c": 30}}', '$.b') |
+------------------------------------------------+
|                                              1 |
```

```
+---------------------------------------------+
```

- `JSON_TYPE(json_val)`

  Returns a `utf8mb4` string indicating the type of a JSON value. This can be an object, an array, or a scalar type, as shown here:

```
mysql> SET @j = '{"a": [10, true]}';
mysql> SELECT JSON_TYPE(@j);
+---------------+
| JSON_TYPE(@j) |
+---------------+
| OBJECT        |
+---------------+
mysql> SELECT JSON_TYPE(JSON_EXTRACT(@j, '$.a'));
+-----------------------------------+
| JSON_TYPE(JSON_EXTRACT(@j, '$.a')) |
+-----------------------------------+
| ARRAY                             |
+-----------------------------------+
mysql> SELECT JSON_TYPE(JSON_EXTRACT(@j, '$.a[0]'));
+--------------------------------------+
| JSON_TYPE(JSON_EXTRACT(@j, '$.a[0]')) |
+--------------------------------------+
| INTEGER                              |
+--------------------------------------+
mysql> SELECT JSON_TYPE(JSON_EXTRACT(@j, '$.a[1]'));
+--------------------------------------+
| JSON_TYPE(JSON_EXTRACT(@j, '$.a[1]')) |
+--------------------------------------+
| BOOLEAN                              |
+--------------------------------------+
```

  `JSON_TYPE()` returns `NULL` if the argument is `NULL`:

```
mysql> SELECT JSON_TYPE(NULL);
+-----------------+
| JSON_TYPE(NULL) |
+-----------------+
| NULL            |
+-----------------+
```

  An error occurs if the argument is not a valid JSON value:

```
mysql> SELECT JSON_TYPE(1);
ERROR 3146 (22032): Invalid data type for JSON data in argument 1
to function json_type; a JSON string or JSON type is required.
```

  For a non-`NULL`, non-error result, the following list describes the possible `JSON_TYPE()` return values:

  - Purely JSON types:
    - `OBJECT`: JSON objects
    - `ARRAY`: JSON arrays
    - `BOOLEAN`: The JSON true and false literals
    - `NULL`: The JSON null literal
  - Numeric types:
    - `INTEGER`: MySQL `TINYINT`, `SMALLINT`, `MEDIUMINT` and `INT` and `BIGINT` scalars
    - `DOUBLE`: MySQL `DOUBLE` `FLOAT` scalars
    - `DECIMAL`: MySQL `DECIMAL` and `NUMERIC` scalars

  - Temporal types:
    - `DATETIME`: MySQL `DATETIME` and `TIMESTAMP` scalars
    - `DATE`: MySQL `DATE` scalars
    - `TIME`: MySQL `TIME` scalars
  - String types:
    - `STRING`: MySQL `utf8mb3` character type scalars: `CHAR`, `VARCHAR`, `TEXT`, `ENUM`, and `SET`
  - Binary types:
    - `BLOB`: MySQL binary type scalars including `BINARY`, `VARBINARY`, `BLOB`, and `BIT`
  - All other types:
    - `OPAQUE` (raw bits)

- `JSON_VALID(val)`

  Returns 0 or 1 to indicate whether a value is valid JSON. Returns `NULL` if the argument is `NULL`.

```
mysql> SELECT JSON_VALID('{"a": 1}');
+-----------------------+
| JSON_VALID('{"a": 1}') |
+-----------------------+
|                     1 |
+-----------------------+
mysql> SELECT JSON_VALID('hello'), JSON_VALID('"hello"');
+--------------------+-----------------------+
| JSON_VALID('hello') | JSON_VALID('"hello"') |
+--------------------+-----------------------+
|                  0 |                     1 |
+--------------------+-----------------------+
```

## 12.18.6 JSON Table Functions

This section contains information about JSON functions that convert JSON data to tabular data. MySQL 8.0 supports one such function, `JSON_TABLE()`.

`JSON_TABLE(expr, path COLUMNS (column_list) [AS] alias)`

Extracts data from a JSON document and returns it as a relational table having the specified columns. The complete syntax for this function is shown here:

```
JSON_TABLE(
    expr,
    path COLUMNS (column_list)
)   [AS] alias

column_list:
    column[, column][, ...]

column:
    name FOR ORDINALITY
    |  name type PATH string path [on_empty] [on_error]
    |  name type EXISTS PATH string path
    |  NESTED [PATH] path COLUMNS (column_list)

on_empty:
    {NULL | DEFAULT json_string | ERROR} ON EMPTY

on_error:
    {NULL | DEFAULT json_string | ERROR} ON ERROR
```

*expr*: This is an expression that returns JSON data. This can be a constant (`'{"a":1}'`), a column (`t1.json_data`, given table `t1` specified prior to `JSON_TABLE()` in the `FROM` clause), or a function call (`JSON_EXTRACT(t1.json_data,'$.post.comments')`).

*path*: A JSON path expression, which is applied to the data source. We refer to the JSON value matching the path as the *row source*; this is used to generate a row of relational data. The `COLUMNS` clause evaluates the row source, finds specific JSON values within the row source, and returns those JSON values as SQL values in individual columns of a row of relational data.

The *alias* is required. The usual rules for table aliases apply (see Section 9.2, "Schema Object Names").

Beginning with MySQL 8.0.27, this function compares column names in case-insensitive fashion.

`JSON_TABLE()` supports four types of columns, described in the following list:

1. *name* `FOR ORDINALITY`: This type enumerates rows in the `COLUMNS` clause; the column named *name* is a counter whose type is `UNSIGNED INT`, and whose initial value is 1. This is equivalent to specifying a column as `AUTO_INCREMENT` in a `CREATE TABLE` statement, and can be used to distinguish parent rows with the same value for multiple rows generated by a `NESTED [PATH]` clause.

2. *name type* `PATH` *string_path* `[`*on_empty*`] [`*on_error*`]`: Columns of this type are used to extract values specified by *string_path*. *type* is a MySQL scalar data type (that is, it cannot be an object or array). `JSON_TABLE()` extracts data as JSON then coerces it to the column type, using the regular automatic type conversion applying to JSON data in MySQL. A missing value triggers the *on_empty* clause. Saving an object or array triggers the optional *on error* clause; this also occurs when an error takes place during coercion from the value saved as JSON to the table column, such as trying to save the string `'asd'` to an integer column.

3. *name type* `EXISTS PATH` *path*: This column returns 1 if any data is present at the location specified by *path*, and 0 otherwise. *type* can be any valid MySQL data type, but should normally be specified as some variety of `INT`.

4. `NESTED [PATH]` *path* `COLUMNS (`*column_list*`)`: This flattens nested objects or arrays in JSON data into a single row along with the JSON values from the parent object or array. Using multiple `PATH` options allows projection of JSON values from multiple levels of nesting into a single row.

   The *path* is relative to the parent path row path of `JSON_TABLE()`, or the path of the parent `NESTED [PATH]` clause in the event of nested paths.

*on empty*, if specified, determines what `JSON_TABLE()` does in the event that data is missing (depending on type). This clause is also triggered on a column in a `NESTED PATH` clause when the latter has no match and a `NULL` complemented row is produced for it. *on empty* takes one of the following values:

- `NULL ON EMPTY`: The column is set to `NULL`; this is the default behavior.
- `DEFAULT` *json_string* `ON EMPTY`: the provided *json_string* is parsed as JSON, as long as it is valid, and stored instead of the missing value. Column type rules also apply to the default value.
- `ERROR ON EMPTY`: An error is thrown.

If used, *on_error* takes one of the following values with the corresponding result as shown here:

- `NULL ON ERROR`: The column is set to `NULL`; this is the default behavior.
- `DEFAULT` *json string* `ON ERROR`: The *json_string* is parsed as JSON (provided that it is valid) and stored instead of the object or array.
- `ERROR ON ERROR`: An error is thrown.

Prior to MySQL 8.0.20, a warning was thrown if a type conversion error occurred with `NULL ON ERROR` or `DEFAULT ... ON ERROR` was specified or implied. In MySQL 8.0.20 and later, this is no longer the case. (Bug #30628330)

Previously, it was possible to specify `ON EMPTY` and `ON ERROR` clauses in either order. This runs counter to the SQL standard, which stipulates that `ON EMPTY`, if specified, must precede any `ON ERROR` clause. For this reason, beginning with MySQL 8.0.20, specifying `ON ERROR` before `ON EMPTY` is deprecated; trying to do so causes the server to issue a warning. Expect support for the nonstandard syntax to be removed in a future version of MySQL.

When a value saved to a column is truncated, such as saving 3.14159 in a `DECIMAL(10,1)` column, a warning is issued independently of any `ON ERROR` option. When multiple values are truncated in a single statement, the warning is issued only once.

Prior to MySQL 8.0.21, when the expression and path passed to this function resolved to JSON null, `JSON_TABLE()` raised an error. In MySQL 8.0.21 and later, it returns SQL `NULL` in such cases, in accordance with the SQL standard, as shown here (Bug #31345503, Bug #99557):

```
mysql> SELECT *
    ->   FROM
    ->     JSON_TABLE(
    ->       '[ {"c1": null} ]',
    ->       '$[*]' COLUMNS( c1 INT PATH '$.c1' ERROR ON ERROR )
    ->     ) as jt;
+------+
| c1   |
+------+
| NULL |
+------+
1 row in set (0.00 sec)
```

The following query demonstrates the use of `ON EMPTY` and `ON ERROR`. The row corresponding to `{"b":1}` is empty for the path `"$.a"`, and attempting to save `[1,2]` as a scalar produces an error; these rows are highlighted in the output shown.

```
mysql> SELECT *
    -> FROM
    ->   JSON_TABLE(
    ->     '[{"a":"3"},{"a":2},{"b":1},{"a":0},{"a":[1,2]}]',
    ->     "$[*]"
    ->     COLUMNS(
    ->       rowid FOR ORDINALITY,
    ->       ac VARCHAR(100) PATH "$.a" DEFAULT '111' ON EMPTY DEFAULT '999' ON ERROR,
    ->       aj JSON PATH "$.a" DEFAULT '{"x": 333}' ON EMPTY,
    ->       bx INT EXISTS PATH "$.b"
    ->     )
    ->   ) AS tt;

+-------+------+------------+------+
| rowid | ac   | aj         | bx   |
+-------+------+------------+------+
|     1 | 3    | "3"        |    0 |
|     2 | 2    | 2          |    0 |
|     3 | 111  | {"x": 333} |    1 |
|     4 | 0    | 0          |    0 |
|     5 | 999  | [1, 2]     |    0 |
+-------+------+------------+------+
5 rows in set (0.00 sec)
```

Column names are subject to the usual rules and limitations governing table column names. See Section 9.2, “Schema Object Names”.

All JSON and JSON path expressions are checked for validity; an invalid expression of either type causes an error.

Each match for the *`path`* preceding the `COLUMNS` keyword maps to an individual row in the result table. For example, the following query gives the result shown here:

```
mysql> SELECT *
    -> FROM
    ->   JSON_TABLE(
    ->     '[{"x":2,"y":"8"},{"x":"3","y":"7"},{"x":"4","y":6}]',
    ->     "$[*]" COLUMNS(
    ->       xval VARCHAR(100) PATH "$.x",
    ->       yval VARCHAR(100) PATH "$.y"
    ->     )
    ->   ) AS  jt1;

+------+------+
| xval | yval |
+------+------+
| 2    | 8    |
| 3    | 7    |
| 4    | 6    |
+------+------+
```

The expression `"$[*]"` matches each element of the array. You can filter the rows in the result by modifying the path. For example, using `"$[1]"` limits extraction to the second element of the JSON array used as the source, as shown here:

```
mysql> SELECT *
    -> FROM
    ->   JSON_TABLE(
    ->     '[{"x":2,"y":"8"},{"x":"3","y":"7"},{"x":"4","y":6}]',
    ->     "$[1]" COLUMNS(
    ->       xval VARCHAR(100) PATH "$.x",
    ->       yval VARCHAR(100) PATH "$.y"
    ->     )
    ->   ) AS  jt1;

+------+------+
| xval | yval |
+------+------+
| 3    | 7    |
+------+------+
```

Within a column definition, `"$"` passes the entire match to the column; `"$.x"` and `"$.y"` pass only the values corresponding to the keys `x` and `y`, respectively, within that match. For more information, see JSON Path Syntax.

`NESTED PATH` (or simply `NESTED`; `PATH` is optional) produces a set of records for each match in the `COLUMNS` clause to which it belongs. If there is no match, all columns of the nested path are set to `NULL`. This implements an outer join between the topmost clause and `NESTED [PATH]`. An inner join can be emulated by applying a suitable condition in the `WHERE` clause, as shown here:

```
mysql> SELECT *
    -> FROM
    ->   JSON_TABLE(
    ->     '[ {"a": 1, "b": [11,111]}, {"a": 2, "b": [22,222]}, {"a":3}]',
    ->     '$[*]' COLUMNS(
    ->             a INT PATH '$.a',
    ->             NESTED PATH '$.b[*]' COLUMNS (b INT PATH '$')
    ->            )
    ->    ) AS jt
    -> WHERE b IS NOT NULL;

+------+------+
| a    | b    |
+------+------+
|    1 |   11 |
|    1 |  111 |
|    2 |   22 |
|    2 |  222 |
+------+------+
```

Sibling nested paths—that is, two or more instances of `NESTED [PATH]` in the same `COLUMNS` clause—are processed one after another, one at a time. While one nested path is producing records, columns

of any sibling nested path expressions are set to NULL. This means that the total number of records for a single match within a single containing COLUMNS clause is the sum and not the product of all records produced by NESTED [PATH] modifiers, as shown here:

```
mysql> SELECT *
    -> FROM
    ->   JSON_TABLE(
    ->     '[{"a": 1, "b": [11,111]}, {"a": 2, "b": [22,222]}]',
    ->     '$[*]' COLUMNS(
    ->         a INT PATH '$.a',
    ->         NESTED PATH '$.b[*]' COLUMNS (b1 INT PATH '$'),
    ->         NESTED PATH '$.b[*]' COLUMNS (b2 INT PATH '$')
    ->     )
    -> ) AS jt;

+------+------+------+
| a    | b1   | b2   |
+------+------+------+
|    1 |   11 | NULL |
|    1 |  111 | NULL |
|    1 | NULL |   11 |
|    1 | NULL |  111 |
|    2 |   22 | NULL |
|    2 |  222 | NULL |
|    2 | NULL |   22 |
|    2 | NULL |  222 |
+------+------+------+
```

A FOR ORDINALITY column enumerates records produced by the COLUMNS clause, and can be used to distinguish parent records of a nested path, especially if values in parent records are the same, as can be seen here:

```
mysql> SELECT *
    -> FROM
    ->   JSON_TABLE(
    ->     '[{"a": "a_val",
    '>       "b": [{"c": "c_val", "l": [1,2]}]},
    '>     {"a": "a_val",
    '>       "b": [{"c": "c_val","l": [11]}, {"c": "c_val", "l": [22]}]}]',
    ->     '$[*]' COLUMNS(
    ->       top_ord FOR ORDINALITY,
    ->       apath VARCHAR(10) PATH '$.a',
    ->       NESTED PATH '$.b[*]' COLUMNS (
    ->         bpath VARCHAR(10) PATH '$.c',
    ->         ord FOR ORDINALITY,
    ->         NESTED PATH '$.l[*]' COLUMNS (lpath varchar(10) PATH '$')
    ->         )
    ->     )
    -> ) as jt;

+---------+---------+---------+------+-------+
| top_ord | apath   | bpath   | ord  | lpath |
+---------+---------+---------+------+-------+
|       1 |  a_val  |  c_val  |    1 | 1     |
|       1 |  a_val  |  c_val  |    1 | 2     |
|       2 |  a_val  |  c_val  |    1 | 11    |
|       2 |  a_val  |  c_val  |    2 | 22    |
+---------+---------+---------+------+-------+
```

The source document contains an array of two elements; each of these elements produces two rows. The values of apath and bpath are the same over the entire result set; this means that they cannot be used to determine whether lpath values came from the same or different parents. The value of the ord column remains the same as the set of records having top_ord equal to 1, so these two values are from a single object. The remaining two values are from different objects, since they have different values in the ord column.

Normally, you cannot join a derived table which depends on columns of preceding tables in the same FROM clause. MySQL, per the SQL standard, makes an exception for table functions; these are considered lateral derived tables, even in versions of MySQL that do not yet support the LATERAL

keyword (8.0.13 and earlier). In versions where `LATERAL` is supported (8.0.14 and later), it is implicit, and for this reason is not allowed before `JSON_TABLE()`, also according to the standard.

Suppose you have a table `t1` created and populated using the statements shown here:

```
CREATE TABLE t1 (c1 INT, c2 CHAR(1), c3 JSON);

INSERT INTO t1 () VALUES
 ROW(1, 'z', JSON_OBJECT('a', 23, 'b', 27, 'c', 1)),
 ROW(1, 'y', JSON_OBJECT('a', 44, 'b', 22, 'c', 11)),
 ROW(2, 'x', JSON_OBJECT('b', 1, 'c', 15)),
 ROW(3, 'w', JSON_OBJECT('a', 5, 'b', 6, 'c', 7)),
 ROW(5, 'v', JSON_OBJECT('a', 123, 'c', 1111))
;
```

You can then execute joins, such as this one, in which `JSON_TABLE()` acts as a derived table while at the same time it refers to a column in a previously referenced table:

```
SELECT c1, c2, JSON_EXTRACT(c3, '$.*')
FROM t1 AS m
JOIN
JSON_TABLE(
  m.c3,
  '$.*'
  COLUMNS(
    at VARCHAR(10) PATH '$.a' DEFAULT '1' ON EMPTY,
    bt VARCHAR(10) PATH '$.b' DEFAULT '2' ON EMPTY,
    ct VARCHAR(10) PATH '$.c' DEFAULT '3' ON EMPTY
  )
) AS tt
ON m.c1 > tt.at;
```

Attempting to use the `LATERAL` keyword with this query raises `ER_PARSE_ERROR`.

## 12.18.7 JSON Schema Validation Functions

Beginning with MySQL 8.0.17, MySQL supports validation of JSON documents against JSON schemas conforming to Draft 4 of the JSON Schema specification. This can be done using either of the functions detailed in this section, both of which take two arguments, a JSON schema, and a JSON document which is validated against the schema. `JSON_SCHEMA_VALID()` returns true if the document validates against the schema, and false if it does not; `JSON_SCHEMA_VALIDATION_REPORT()` provides a report in JSON format on the validation.

Both functions handle null or invalid input as follows:

- If at least one of the arguments is `NULL`, the function returns `NULL`.
- If at least one of the arguments is not valid JSON, the function raises an error (`ER_INVALID_TYPE_FOR_JSON`)
- In addition, if the schema is not a valid JSON object, the function returns `ER_INVALID_JSON_TYPE`.

MySQL supports the `required` attribute in JSON schemas to enforce the inclusion of required properties (see the examples in the function descriptions).

MySQL supports the `id`, `$schema`, `description`, and `type` attributes in JSON schemas but does not require any of these.

MySQL does not support external resources in JSON schemas; using the `$ref` keyword causes `JSON_SCHEMA_VALID()` to fail with `ER_NOT_SUPPORTED_YET`.

**Note**

MySQL supports regular expression patterns in JSON schema, which supports but silently ignores invalid patterns (see the description of `JSON_SCHEMA_VALID()` for an example).

These functions are described in detail in the following list:

- `JSON_SCHEMA_VALID(schema,document)`

  Validates a JSON *`document`* against a JSON *`schema`*. Both *`schema`* and *`document`* are required. The schema must be a valid JSON object; the document must be a valid JSON document. Provided that these conditions are met: If the document validates against the schema, the function returns true (1); otherwise, it returns false (0).

  In this example, we set a user variable `@schema` to the value of a JSON schema for geographical coordinates, and another one `@document` to the value of a JSON document containing one such coordinate. We then verify that `@document` validates according to `@schema` by using them as the arguments to `JSON_SCHEMA_VALID()`:

```
mysql> SET @schema = '{
    '>  "id": "http://json-schema.org/geo",
    '> "$schema": "http://json-schema.org/draft-04/schema#",
    '> "description": "A geographical coordinate",
    '> "type": "object",
    '> "properties": {
    '>   "latitude": {
    '>     "type": "number",
    '>     "minimum": -90,
    '>     "maximum": 90
    '>   },
    '>   "longitude": {
    '>     "type": "number",
    '>     "minimum": -180,
    '>     "maximum": 180
    '>   }
    '> },
    '> "required": ["latitude", "longitude"]
    '>}';
Query OK, 0 rows affected (0.01 sec)

mysql> SET @document = '{
    '> "latitude": 63.444697,
    '> "longitude": 10.445118
    '>}';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT JSON_SCHEMA_VALID(@schema, @document);
+--------------------------------------+
| JSON_SCHEMA_VALID(@schema, @document) |
+--------------------------------------+
|                                    1 |
+--------------------------------------+
1 row in set (0.00 sec)
```

  Since `@schema` contains the `required` attribute, we can set `@document` to a value that is otherwise valid but does not contain the required properties, then test it against `@schema`, like this:

```
mysql> SET @document = '{}';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT JSON_SCHEMA_VALID(@schema, @document);
+--------------------------------------+
| JSON_SCHEMA_VALID(@schema, @document) |
+--------------------------------------+
|                                    0 |
+--------------------------------------+
1 row in set (0.00 sec)
```

  If we now set the value of `@schema` to the same JSON schema but without the `required` attribute, `@document` validates because it is a valid JSON object, even though it contains no properties, as shown here:

```
mysql> SET @schema = '{
```

```
    '> "id": "http://json-schema.org/geo",
    '> "$schema": "http://json-schema.org/draft-04/schema#",
    '> "description": "A geographical coordinate",
    '> "type": "object",
    '> "properties": {
    '>   "latitude": {
    '>     "type": "number",
    '>     "minimum": -90,
    '>     "maximum": 90
    '>   },
    '>   "longitude": {
    '>     "type": "number",
    '>     "minimum": -180,
    '>     "maximum": 180
    '>   }
    '> }
    '>}';
Query OK, 0 rows affected (0.00 sec)


mysql> SELECT JSON_SCHEMA_VALID(@schema, @document);
+--------------------------------------+
| JSON_SCHEMA_VALID(@schema, @document) |
+--------------------------------------+
|                                    1 |
+--------------------------------------+
1 row in set (0.00 sec)
```

**JSON_SCHEMA_VALID() and CHECK constraints.** `JSON_SCHEMA_VALID()` can also be used to enforce `CHECK` constraints.

Consider the table `geo` created as shown here, with a JSON column `coordinate` representing a point of latitude and longitude on a map, governed by the JSON schema used as an argument in a `JSON_SCHEMA_VALID()` call which is passed as the expression for a `CHECK` constraint on this table:

```
mysql> CREATE TABLE geo (
    ->     coordinate JSON,
    ->     CHECK(
    ->         JSON_SCHEMA_VALID(
    ->             '{
    '>                 "type":"object",
    '>                 "properties":{
    '>                       "latitude":{"type":"number", "minimum":-90, "maximum":90},
    '>                       "longitude":{"type":"number", "minimum":-180, "maximum":180}
    '>                 },
    '>                 "required": ["latitude", "longitude"]
    '>             }',
    ->             coordinate
    ->         )
    ->     )
    -> );
Query OK, 0 rows affected (0.45 sec)
```

**Note**

Because a MySQL `CHECK` constraint cannot contain references to variables, you must pass the JSON schema to `JSON_SCHEMA_VALID()` inline when using it to specify such a constraint for a table.

We assign JSON values representing coordinates to three variables, as shown here:

```
mysql> SET @point1 = '{"latitude":59, "longitude":18}';
Query OK, 0 rows affected (0.00 sec)

mysql> SET @point2 = '{"latitude":91, "longitude":0}';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SET @point3 = '{"longitude":120}';
Query OK, 0 rows affected (0.00 sec)
```

The first of these values is valid, as can be seen in the following `INSERT` statement:

```
mysql> INSERT INTO geo VALUES(@point1);
Query OK, 1 row affected (0.05 sec)
```

The second JSON value is invalid and so fails the constraint, as shown here:

```
mysql> INSERT INTO geo VALUES(@point2);
ERROR 3819 (HY000): Check constraint 'geo_chk_1' is violated.
```

In MySQL 8.0.19 and later, you can obtain precise information about the nature of the failure—in this case, that the `latitude` value exceeds the maximum defined in the schema—by issuing a `SHOW WARNINGS` statement:

```
mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Error
   Code: 3934
Message: The JSON document location '#/latitude' failed requirement 'maximum' at
JSON Schema location '#/properties/latitude'.
*************************** 2. row ***************************
  Level: Error
   Code: 3819
Message: Check constraint 'geo_chk_1' is violated.
2 rows in set (0.00 sec)
```

The third coordinate value defined above is also invalid, since it is missing the required `latitude` property. As before, you can see this by attempting to insert the value into the `geo` table, then issuing `SHOW WARNINGS` afterwards:

```
mysql> INSERT INTO geo VALUES(@point3);
ERROR 3819 (HY000): Check constraint 'geo_chk_1' is violated.
mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Error
   Code: 3934
Message: The JSON document location '#' failed requirement 'required' at JSON
Schema location '#'.
*************************** 2. row ***************************
  Level: Error
   Code: 3819
Message: Check constraint 'geo_chk_1' is violated.
2 rows in set (0.00 sec)
```

See Section 13.1.20.6, “CHECK Constraints”, for more information.

JSON Schema has support for specifying regular expression patterns for strings, but the implementation used by MySQL silently ignores invalid patterns. This means that `JSON_SCHEMA_VALID()` can return true even when a regular expression pattern is invalid, as shown here:

```
mysql> SELECT JSON_SCHEMA_VALID('{"type":"string","pattern":"("}', '"abc"');
+-------------------------------------------------------------+
| JSON_SCHEMA_VALID('{"type":"string","pattern":"("}', '"abc"') |
+-------------------------------------------------------------+
|                                                           1 |
+-------------------------------------------------------------+
1 row in set (0.04 sec)
```

- `JSON_SCHEMA_VALIDATION_REPORT(`*`schema`*`,`*`document`*`)`

  Validates a JSON *`document`* against a JSON *`schema`*. Both *`schema`* and *`document`* are required. As with JSON_VALID_SCHEMA(), the schema must be a valid JSON object, and the document must be a valid JSON document. Provided that these conditions are met, the function returns a

report, as a JSON document, on the outcome of the validation. If the JSON document is considered valid according to the JSON Schema, the function returns a JSON object with one property `valid` having the value "true". If the JSON document fails validation, the function returns a JSON object which includes the properties listed here:

- `valid`: Always "false" for a failed schema validation
- `reason`: A human-readable string containing the reason for the failure
- `schema-location`: A JSON pointer URI fragment identifier indicating where in the JSON schema the validation failed (see Note following this list)
- `document-location`: A JSON pointer URI fragment identifier indicating where in the JSON document the validation failed (see Note following this list)
- `schema-failed-keyword`: A string containing the name of the keyword or property in the JSON schema that was violated

**Note**

JSON pointer URI fragment identifiers are defined in RFC 6901 - JavaScript Object Notation (JSON) Pointer. (These are *not* the same as the JSON path notation used by `JSON_EXTRACT()` and other MySQL JSON functions.) In this notation, `#` represents the entire document, and `#/myprop` represents the portion of the document included in the top-level property named `myprop`. See the specification just cited and the examples shown later in this section for more information.

In this example, we set a user variable `@schema` to the value of a JSON schema for geographical coordinates, and another one `@document` to the value of a JSON document containing one such coordinate. We then verify that `@document` validates according to `@schema` by using them as the arguments to `JSON_SCHEMA_VALIDATION_REORT()`:

```
mysql> SET @schema = '{
    '>  "id": "http://json-schema.org/geo",
    '> "$schema": "http://json-schema.org/draft-04/schema#",
    '> "description": "A geographical coordinate",
    '> "type": "object",
    '> "properties": {
    '>   "latitude": {
    '>     "type": "number",
    '>     "minimum": -90,
    '>     "maximum": 90
    '>   },
    '>   "longitude": {
    '>     "type": "number",
    '>     "minimum": -180,
    '>     "maximum": 180
    '>   }
    '> },
    '> "required": ["latitude", "longitude"]
    '>}';
Query OK, 0 rows affected (0.01 sec)

mysql> SET @document = '{
    '> "latitude": 63.444697,
    '> "longitude": 10.445118
    '>}';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT JSON_SCHEMA_VALIDATION_REPORT(@schema, @document);
+------------------------------------------------+
| JSON_SCHEMA_VALIDATION_REPORT(@schema, @document) |
+------------------------------------------------+
| {"valid": true}                                |
+------------------------------------------------+
```

```
1 row in set (0.00 sec)
```

Now we set `@document` such that it specifies an illegal value for one of its properties, like this:

```
mysql> SET @document = '{
    '> "latitude": 63.444697,
    '> "longitude": 310.445118
    '> }';
```

Validation of `@document` now fails when tested with `JSON_SCHEMA_VALIDATION_REPORT()`. The output from the function call contains detailed information about the failure (with the function wrapped by `JSON_PRETTY()` to provide better formatting), as shown here:

```
mysql> SELECT JSON_PRETTY(JSON_SCHEMA_VALIDATION_REPORT(@schema, @document))\G
*************************** 1. row ***************************
JSON_PRETTY(JSON_SCHEMA_VALIDATION_REPORT(@schema, @document)): {
  "valid": false,
  "reason": "The JSON document location '#/longitude' failed requirement 'maximum' at JSON Schema locatio
  "schema-location": "#/properties/longitude",
  "document-location": "#/longitude",
  "schema-failed-keyword": "maximum"
}
1 row in set (0.00 sec)
```

Since `@schema` contains the `required` attribute, we can set `@document` to a value that is otherwise valid but does not contain the required properties, then test it against `@schema`. The output of `JSON_SCHEMA_VALIDATION_REPORT()` shows that validation fails due to lack of a required element, like this:

```
mysql> SET @document = '{}';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT JSON_PRETTY(JSON_SCHEMA_VALIDATION_REPORT(@schema, @document))\G
*************************** 1. row ***************************
JSON_PRETTY(JSON_SCHEMA_VALIDATION_REPORT(@schema, @document)): {
  "valid": false,
  "reason": "The JSON document location '#' failed requirement 'required' at JSON Schema location '#'",
  "schema-location": "#",
  "document-location": "#",
  "schema-failed-keyword": "required"
}
1 row in set (0.00 sec)
```

If we now set the value of `@schema` to the same JSON schema but without the `required` attribute, `@document` validates because it is a valid JSON object, even though it contains no properties, as shown here:

```
mysql> SET @schema = '{
    '> "id": "http://json-schema.org/geo",
    '> "$schema": "http://json-schema.org/draft-04/schema#",
    '> "description": "A geographical coordinate",
    '> "type": "object",
    '> "properties": {
    '>   "latitude": {
    '>     "type": "number",
    '>     "minimum": -90,
    '>     "maximum": 90
    '>   },
    '>   "longitude": {
    '>     "type": "number",
    '>     "minimum": -180,
    '>     "maximum": 180
    '>   }
    '> }
    '>}';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT JSON_SCHEMA_VALIDATION_REPORT(@schema, @document);
+--------------------------------------------------+
```

```
| JSON_SCHEMA_VALIDATION_REPORT(@schema, @document) |
+---------------------------------------------------+
| {"valid": true}                                   |
+---------------------------------------------------+
1 row in set (0.00 sec)
```

## 12.18.8 JSON Utility Functions

This section documents utility functions that act on JSON values, or strings that can be parsed as JSON values. `JSON_PRETTY()` prints out a JSON value in a format that is easy to read. `JSON_STORAGE_SIZE()` and `JSON_STORAGE_FREE()` show, respectively, the amount of storage space used by a given JSON value and the amount of space remaining in a `JSON` column following a partial update.

- `JSON_PRETTY(`*`json_val`*`)`

  Provides pretty-printing of JSON values similar to that implemented in PHP and by other languages and database systems. The value supplied must be a JSON value or a valid string representation of a JSON value. Extraneous whitespaces and newlines present in this value have no effect on the output. For a `NULL` value, the function returns `NULL`. If the value is not a JSON document, or if it cannot be parsed as one, the function fails with an error.

  Formatting of the output from this function adheres to the following rules:

  - Each array element or object member appears on a separate line, indented by one additional level as compared to its parent.
  - Each level of indentation adds two leading spaces.
  - A comma separating individual array elements or object members is printed before the newline that separates the two elements or members.
  - The key and the value of an object member are separated by a colon followed by a space (`': '`).
  - An empty object or array is printed on a single line. No space is printed between the opening and closing brace.
  - Special characters in string scalars and key names are escaped employing the same rules used by the `JSON_QUOTE()` function.

```
mysql> SELECT JSON_PRETTY('123'); # scalar
+--------------------+
| JSON_PRETTY('123') |
+--------------------+
| 123                |
+--------------------+

mysql> SELECT JSON_PRETTY("[1,3,5]"); # array
+------------------------+
| JSON_PRETTY("[1,3,5]") |
+------------------------+
| [
  1,
  3,
  5
]      |
+------------------------+

mysql> SELECT JSON_PRETTY('{"a":"10","b":"15","x":"25"}'); # object
+--------------------------------------------+
| JSON_PRETTY('{"a":"10","b":"15","x":"25"}') |
+--------------------------------------------+
| {
  "a": "10",
  "b": "15",
  "x": "25"
```

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

mysql> SELECT JSON_PRETTY('["a",1,{"key1":
    '>    "value1"},"5",     "77" ,
    '>       {"key2":["value3","valueX",
    '> "valueY"]},"j", "2"   ]')\G  # nested arrays and objects
*************************** 1. row ***************************
JSON_PRETTY('["a",1,{"key1":
             "value1"},"5",     "77" ,
                {"key2":["value3","valuex",
          "valuey"]},"j", "2"   ]'): [
  "a",
  1,
  {
    "key1": "value1"
  },
  "5",
  "77",
  {
    "key2": [
      "value3",
      "valuex",
      "valuey"
    ]
  },
  "j",
  "2"
]
```

- `JSON_STORAGE_FREE(`*`json_val`*`)`

  For a `JSON` column value, this function shows how much storage space was freed in its binary representation after it was updated in place using `JSON_SET()`, `JSON_REPLACE()`, or `JSON_REMOVE()`. The argument can also be a valid JSON document or a string which can be parsed as one—either as a literal value or as the value of a user variable—in which case the function returns 0. It returns a positive, nonzero value if the argument is a `JSON` column value which has been updated as described previously, such that its binary representation takes up less space than it did prior to the update. For a `JSON` column which has been updated such that its binary representation is the same as or larger than before, or if the update was not able to take advantage of a partial update, it returns 0; it returns `NULL` if the argument is `NULL`.

  If *`json_val`* is not `NULL`, and neither is a valid JSON document nor can be successfully parsed as one, an error results.

  In this example, we create a table containing a `JSON` column, then insert a row containing a JSON object:

```
mysql> CREATE TABLE jtable (jcol JSON);
Query OK, 0 rows affected (0.38 sec)

mysql> INSERT INTO jtable VALUES
    ->     ('{"a": 10, "b": "wxyz", "c": "[true, false]"}');
Query OK, 1 row affected (0.04 sec)

mysql> SELECT * FROM jtable;
+-------------------------------------------+
| jcol                                      |
+-------------------------------------------+
| {"a": 10, "b": "wxyz", "c": "[true, false]"} |
+-------------------------------------------+
1 row in set (0.00 sec)
```

  Now we update the column value using `JSON_SET()` such that a partial update can be performed; in this case, we replace the value pointed to by the `c` key (the array `[true, false]`) with one that takes up less space (the integer `1`):

```
mysql> UPDATE jtable
```

```
    ->     SET jcol = JSON_SET(jcol, "$.a", 10, "$.b", "wxyz", "$.c", 1);
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM jtable;
+--------------------------------+
| jcol                           |
+--------------------------------+
| {"a": 10, "b": "wxyz", "c": 1} |
+--------------------------------+
1 row in set (0.00 sec)

mysql> SELECT JSON_STORAGE_FREE(jcol) FROM jtable;
+-------------------------+
| JSON_STORAGE_FREE(jcol) |
+-------------------------+
|                      14 |
+-------------------------+
1 row in set (0.00 sec)
```

The effects of successive partial updates on this free space are cumulative, as shown in this example using `JSON_SET()` to reduce the space taken up by the value having key `b` (and making no other changes):

```
mysql> UPDATE jtable
    ->     SET jcol = JSON_SET(jcol, "$.a", 10, "$.b", "wx", "$.c", 1);
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT JSON_STORAGE_FREE(jcol) FROM jtable;
+-------------------------+
| JSON_STORAGE_FREE(jcol) |
+-------------------------+
|                      16 |
+-------------------------+
1 row in set (0.00 sec)
```

Updating the column without using `JSON_SET()`, `JSON_REPLACE()`, or `JSON_REMOVE()` means that the optimizer cannot perform the update in place; in this case, `JSON_STORAGE_FREE()` returns 0, as shown here:

```
mysql> UPDATE jtable SET jcol = '{"a": 10, "b": 1}';
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT JSON_STORAGE_FREE(jcol) FROM jtable;
+-------------------------+
| JSON_STORAGE_FREE(jcol) |
+-------------------------+
|                       0 |
+-------------------------+
1 row in set (0.00 sec)
```

Partial updates of JSON documents can be performed only on column values. For a user variable that stores a JSON value, the value is always completely replaced, even when the update is performed using `JSON_SET()`:

```
mysql> SET @j = '{"a": 10, "b": "wxyz", "c": "[true, false]"}';
Query OK, 0 rows affected (0.00 sec)

mysql> SET @j = JSON_SET(@j, '$.a', 10, '$.b', 'wxyz', '$.c', '1');
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @j, JSON_STORAGE_FREE(@j) AS Free;
+---------------------------------+------+
| @j                              | Free |
+---------------------------------+------+
| {"a": 10, "b": "wxyz", "c": "1"} |    0 |
+---------------------------------+------+
```

```
1 row in set (0.00 sec)
```

For a JSON literal, this function always returns 0:

```
mysql> SELECT JSON_STORAGE_FREE('{"a": 10, "b": "wxyz", "c": "1"}') AS Free;
+------+
| Free |
+------+
|    0 |
+------+
1 row in set (0.00 sec)
```

- `JSON_STORAGE_SIZE(json_val)`

  This function returns the number of bytes used to store the binary representation of a JSON document. When the argument is a `JSON` column, this is the space used to store the JSON document as it was inserted into the column, prior to any partial updates that may have been performed on it afterwards. *`json_val`* must be a valid JSON document or a string which can be parsed as one. In the case where it is string, the function returns the amount of storage space in the JSON binary representation that is created by parsing the string as JSON and converting it to binary. It returns `NULL` if the argument is `NULL`.

  An error results when *`json_val`* is not `NULL`, and is not—or cannot be successfully parsed as—a JSON document.

  To illustrate this function's behavior when used with a `JSON` column as its argument, we create a table named `jtable` containing a `JSON` column `jcol`, insert a JSON value into the table, then obtain the storage space used by this column with `JSON_STORAGE_SIZE()`, as shown here:

```
mysql> CREATE TABLE jtable (jcol JSON);
Query OK, 0 rows affected (0.42 sec)

mysql> INSERT INTO jtable VALUES
    ->     ('{"a": 1000, "b": "wxyz", "c": "[1, 3, 5, 7]"}');
Query OK, 1 row affected (0.04 sec)

mysql> SELECT
    ->     jcol,
    ->     JSON_STORAGE_SIZE(jcol) AS Size,
    ->     JSON_STORAGE_FREE(jcol) AS Free
    -> FROM jtable;
+-----------------------------------------------+------+------+
| jcol                                          | Size | Free |
+-----------------------------------------------+------+------+
| {"a": 1000, "b": "wxyz", "c": "[1, 3, 5, 7]"} |   47 |    0 |
+-----------------------------------------------+------+------+
1 row in set (0.00 sec)
```

  According to the output of `JSON_STORAGE_SIZE()`, the JSON document inserted into the column takes up 47 bytes. We also checked the amount of space freed by any previous partial updates of the column using `JSON_STORAGE_FREE()`; since no updates have yet been performed, this is 0, as expected.

  Next we perform an `UPDATE` on the table that should result in a partial update of the document stored in `jcol`, and then test the result as shown here:

```
mysql> UPDATE jtable SET jcol =
    ->     JSON_SET(jcol, "$.b", "a");
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT
    ->     jcol,
    ->     JSON_STORAGE_SIZE(jcol) AS Size,
    ->     JSON_STORAGE_FREE(jcol) AS Free
    -> FROM jtable;
+-----------------------------------------+------+------+
```

```
| jcol                                       | Size | Free |
+--------------------------------------------+------+------+
| {"a": 1000, "b": "a", "c": "[1, 3, 5, 7]"} |   47 |    3 |
+--------------------------------------------+------+------+
1 row in set (0.00 sec)
```

The value returned by `JSON_STORAGE_FREE()` in the previous query indicates that a partial update of the JSON document was performed, and that this freed 3 bytes of space used to store it. The result returned by `JSON_STORAGE_SIZE()` is unchanged by the partial update.

Partial updates are supported for updates using `JSON_SET()`, `JSON_REPLACE()`, or `JSON_REMOVE()`. The direct assignment of a value to a `JSON` column cannot be partially updated; following such an update, `JSON_STORAGE_SIZE()` always shows the storage used for the newly-set value:

```
mysql> UPDATE jtable
mysql>     SET jcol = '{"a": 4.55, "b": "wxyz", "c": "[true, false]"}';
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT
    ->     jcol,
    ->     JSON_STORAGE_SIZE(jcol) AS Size,
    ->     JSON_STORAGE_FREE(jcol) AS Free
    -> FROM jtable;
+-----------------------------------------------+------+------+
| jcol                                          | Size | Free |
+-----------------------------------------------+------+------+
| {"a": 4.55, "b": "wxyz", "c": "[true, false]"} |   56 |    0 |
+-----------------------------------------------+------+------+
1 row in set (0.00 sec)
```

A JSON user variable cannot be partially updated. This means that this function always shows the space currently used to store a JSON document in a user variable:

```
mysql> SET @j = '[100, "sakila", [1, 3, 5], 425.05]';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @j, JSON_STORAGE_SIZE(@j) AS Size;
+-----------------------------------+------+
| @j                                | Size |
+-----------------------------------+------+
| [100, "sakila", [1, 3, 5], 425.05] |   45 |
+-----------------------------------+------+
1 row in set (0.00 sec)

mysql> SET @j = JSON_SET(@j, '$[1]', "json");
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @j, JSON_STORAGE_SIZE(@j) AS Size;
+-----------------------------------+------+
| @j                                | Size |
+-----------------------------------+------+
| [100, "json", [1, 3, 5], 425.05] |   43 |
+-----------------------------------+------+
1 row in set (0.00 sec)

mysql> SET @j = JSON_SET(@j, '$[2][0]', JSON_ARRAY(10, 20, 30));
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @j, JSON_STORAGE_SIZE(@j) AS Size;
+-------------------------------------------+------+
| @j                                        | Size |
+-------------------------------------------+------+
| [100, "json", [[10, 20, 30], 3, 5], 425.05] |   56 |
+-------------------------------------------+------+
```

```
1 row in set (0.00 sec)
```

For a JSON literal, this function always returns the current storage space used:

```
mysql> SELECT
    ->     JSON_STORAGE_SIZE('[100, "sakila", [1, 3, 5], 425.05]') AS A,
    ->     JSON_STORAGE_SIZE('{"a": 1000, "b": "a", "c": "[1, 3, 5, 7]"}') AS B,
    ->     JSON_STORAGE_SIZE('{"a": 1000, "b": "wxyz", "c": "[1, 3, 5, 7]"}') AS C,
    ->     JSON_STORAGE_SIZE('[100, "json", [[10, 20, 30], 3, 5], 425.05]') AS D;
+----+----+----+----+
| A  | B  | C  | D  |
+----+----+----+----+
| 45 | 44 | 47 | 56 |
+----+----+----+----+
1 row in set (0.00 sec)
```

# 12.19 Functions Used with Global Transaction Identifiers (GTIDs)

The functions described in this section are used with GTID-based replication. It is important to keep in mind that all of these functions take string representations of GTID sets as arguments. As such, the GTID sets must always be quoted when used with them. See GTID Sets for more information.

The union of two GTID sets is simply their representations as strings, joined together with an interposed comma. In other words, you can define a very simple function for obtaining the union of two GTID sets, similar to that created here:

```
CREATE FUNCTION GTID_UNION(g1 TEXT, g2 TEXT)
    RETURNS TEXT DETERMINISTIC
    RETURN CONCAT(g1,',',g2);
```

For more information about GTIDs and how these GTID functions are used in practice, see Section 17.1.3, “Replication with Global Transaction Identifiers”.

**Table 12.24 GTID Functions**

| Name | Description | Deprecated |
|---|---|---|
| `GTID_SUBSET()` | Return true if all GTIDs in subset are also in set; otherwise false. | |
| `GTID_SUBTRACT()` | Return all GTIDs in set that are not in subset. | |
| `WAIT_FOR_EXECUTED_GTID_SET()` | Wait until the given GTIDs have executed on the replica. | |
| `WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS()` | Use `WAIT_FOR_EXECUTED_GTID_SET()`. | 8.0.18 |

- `GTID_SUBSET(`*`set1`*`,`*`set2`*`)`

  Given two sets of global transaction identifiers *`set1`* and *`set2`*, returns true if all GTIDs in *`set1`* are also in *`set2`*. Returns `NULL` if *`set1`* or *`set2`* is `NULL`. Returns false otherwise.

  The GTID sets used with this function are represented as strings, as shown in the following examples:

```
mysql> SELECT GTID_SUBSET('3E11FA47-71CA-11E1-9E33-C80AA9429562:23',
    ->     '3E11FA47-71CA-11E1-9E33-C80AA9429562:21-57')\G
*************************** 1. row ***************************
GTID_SUBSET('3E11FA47-71CA-11E1-9E33-C80AA9429562:23',
    '3E11FA47-71CA-11E1-9E33-C80AA9429562:21-57'): 1
1 row in set (0.00 sec)

mysql> SELECT GTID_SUBSET('3E11FA47-71CA-11E1-9E33-C80AA9429562:23-25',
```

```
    ->     '3E11FA47-71CA-11E1-9E33-C80AA9429562:21-57')\G
*************************** 1. row ***************************
GTID_SUBSET('3E11FA47-71CA-11E1-9E33-C80AA9429562:23-25',
    '3E11FA47-71CA-11E1-9E33-C80AA9429562:21-57'): 1
1 row in set (0.00 sec)

mysql> SELECT GTID_SUBSET('3E11FA47-71CA-11E1-9E33-C80AA9429562:20-25',
    ->     '3E11FA47-71CA-11E1-9E33-C80AA9429562:21-57')\G
*************************** 1. row ***************************
GTID_SUBSET('3E11FA47-71CA-11E1-9E33-C80AA9429562:20-25',
    '3E11FA47-71CA-11E1-9E33-C80AA9429562:21-57'): 0
1 row in set (0.00 sec)
```

- `GTID_SUBTRACT(set1,set2)`

  Given two sets of global transaction identifiers *`set1`* and *`set2`*, returns only those GTIDs from *`set1`* that are not in *`set2`*. Returns `NULL` if *`set1`* or *`set2`* is `NULL`.

  All GTID sets used with this function are represented as strings and must be quoted, as shown in these examples:

```
mysql> SELECT GTID_SUBTRACT('3E11FA47-71CA-11E1-9E33-C80AA9429562:21-57',
    ->     '3E11FA47-71CA-11E1-9E33-C80AA9429562:21')\G
*************************** 1. row ***************************
GTID_SUBTRACT('3E11FA47-71CA-11E1-9E33-C80AA9429562:21-57',
    '3E11FA47-71CA-11E1-9E33-C80AA9429562:21'): 3e11fa47-71ca-11e1-9e33-c80aa9429562:22-57
1 row in set (0.00 sec)

mysql> SELECT GTID_SUBTRACT('3E11FA47-71CA-11E1-9E33-C80AA9429562:21-57',
    ->     '3E11FA47-71CA-11E1-9E33-C80AA9429562:20-25')\G
*************************** 1. row ***************************
GTID_SUBTRACT('3E11FA47-71CA-11E1-9E33-C80AA9429562:21-57',
    '3E11FA47-71CA-11E1-9E33-C80AA9429562:20-25'): 3e11fa47-71ca-11e1-9e33-c80aa9429562:26-57
1 row in set (0.00 sec)

mysql> SELECT GTID_SUBTRACT('3E11FA47-71CA-11E1-9E33-C80AA9429562:21-57',
    ->     '3E11FA47-71CA-11E1-9E33-C80AA9429562:23-24')\G
*************************** 1. row ***************************
GTID_SUBTRACT('3E11FA47-71CA-11E1-9E33-C80AA9429562:21-57',
    '3E11FA47-71CA-11E1-9E33-C80AA9429562:23-24'): 3e11fa47-71ca-11e1-9e33-c80aa9429562:21-22:25-57
1 row in set (0.01 sec)
```

- `WAIT_FOR_EXECUTED_GTID_SET(gtid_set[, timeout])`

  Wait until the server has applied all of the transactions whose global transaction identifiers are contained in *`gtid_set`*; that is, until the condition GTID_SUBSET(*`gtid_subset`*, `@@GLOBAL.gtid_executed`) holds. See Section 17.1.3.1, “GTID Format and Storage” for a definition of GTID sets.

  If a timeout is specified, and *`timeout`* seconds elapse before all of the transactions in the GTID set have been applied, the function stops waiting. *`timeout`* is optional, and the default timeout is 0 seconds, in which case the function always waits until all of the transactions in the GTID set have been applied.

  `WAIT_FOR_EXECUTED_GTID_SET()` monitors all the GTIDs that are applied on the server, including transactions that arrive from all replication channels and user clients. It does not take into account whether replication channels have been started or stopped.

  For more information, see Section 17.1.3, “Replication with Global Transaction Identifiers”.

  GTID sets used with this function are represented as strings and so must be quoted as shown in the following example:

```
mysql> SELECT WAIT_FOR_EXECUTED_GTID_SET('3E11FA47-71CA-11E1-9E33-C80AA9429562:1-5');
       -> 0
```

  For a syntax description for GTID sets, see Section 17.1.3.1, “GTID Format and Storage”.

For `WAIT_FOR_EXECUTED_GTID_SET()`, the return value is the state of the query, where 0 represents success, and 1 represents timeout. Any other failures generate an error.

`gtid_mode` cannot be changed to OFF while any client is using this function to wait for GTIDs to be applied.

- `WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS(gtid_set[, timeout][,channel])`

  `WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS()` is deprecated. Use `WAIT_FOR_EXECUTED_GTID_SET()` instead, which works regardless of the replication channel or user client through which the specified transactions arrive on the server.

# 12.20 Aggregate Functions

Aggregate functions operate on sets of values. They are often used with a `GROUP BY` clause to group values into subsets. This section describes most aggregate functions. For information about aggregate functions that operate on geometry values, see Section 12.17.12, “Spatial Aggregate Functions”.

## 12.20.1 Aggregate Function Descriptions

This section describes aggregate functions that operate on sets of values. They are often used with a `GROUP BY` clause to group values into subsets.

**Table 12.25 Aggregate Functions**

| Name | Description |
|---|---|
| `AVG()` | Return the average value of the argument |
| `BIT_AND()` | Return bitwise AND |
| `BIT_OR()` | Return bitwise OR |
| `BIT_XOR()` | Return bitwise XOR |
| `COUNT()` | Return a count of the number of rows returned |
| `COUNT(DISTINCT)` | Return the count of a number of different values |
| `GROUP_CONCAT()` | Return a concatenated string |
| `JSON_ARRAYAGG()` | Return result set as a single JSON array |
| `JSON_OBJECTAGG()` | Return result set as a single JSON object |
| `MAX()` | Return the maximum value |
| `MIN()` | Return the minimum value |
| `STD()` | Return the population standard deviation |
| `STDDEV()` | Return the population standard deviation |
| `STDDEV_POP()` | Return the population standard deviation |
| `STDDEV_SAMP()` | Return the sample standard deviation |
| `SUM()` | Return the sum |
| `VAR_POP()` | Return the population standard variance |
| `VAR_SAMP()` | Return the sample variance |
| `VARIANCE()` | Return the population standard variance |

Unless otherwise stated, aggregate functions ignore `NULL` values.

If you use an aggregate function in a statement containing no `GROUP BY` clause, it is equivalent to grouping on all rows. For more information, see Section 12.20.3, “MySQL Handling of GROUP BY”.

Most aggregate functions can be used as window functions. Those that can be used this way are signified in their syntax description by `[over_clause]`, representing an optional `OVER` clause.

*over_clause* is described in Section 12.21.2, “Window Function Concepts and Syntax”, which also includes other information about window function usage.

For numeric arguments, the variance and standard deviation functions return a DOUBLE value. The SUM() and AVG() functions return a DECIMAL value for exact-value arguments (integer or DECIMAL), and a DOUBLE value for approximate-value arguments (FLOAT or DOUBLE).

The SUM() and AVG() aggregate functions do not work with temporal values. (They convert the values to numbers, losing everything after the first nonnumeric character.) To work around this problem, convert to numeric units, perform the aggregate operation, and convert back to a temporal value. Examples:

```
SELECT SEC_TO_TIME(SUM(TIME_TO_SEC(time_col))) FROM tbl_name;
SELECT FROM_DAYS(SUM(TO_DAYS(date_col))) FROM tbl_name;
```

Functions such as SUM() or AVG() that expect a numeric argument cast the argument to a number if necessary. For SET or ENUM values, the cast operation causes the underlying numeric value to be used.

The BIT_AND(), BIT_OR(), and BIT_XOR() aggregate functions perform bit operations. Prior to MySQL 8.0, bit functions and operators required BIGINT (64-bit integer) arguments and returned BIGINT values, so they had a maximum range of 64 bits. Non-BIGINT arguments were converted to BIGINT prior to performing the operation and truncation could occur.

In MySQL 8.0, bit functions and operators permit binary string type arguments (BINARY, VARBINARY, and the BLOB types) and return a value of like type, which enables them to take arguments and produce return values larger than 64 bits. For discussion about argument evaluation and result types for bit operations, see the introductory discussion in Section 12.13, “Bit Functions and Operators”.

- AVG([DISTINCT] *expr*) [*over_clause*]

  Returns the average value of *expr*. The DISTINCT option can be used to return the average of the distinct values of *expr*.

  If there are no matching rows, AVG() returns NULL. The function also returns NULL if *expr* is NULL.

  This function executes as a window function if *over_clause* is present. *over_clause* is as described in Section 12.21.2, “Window Function Concepts and Syntax”; it cannot be used with DISTINCT.

  ```
  mysql> SELECT student_name, AVG(test_score)
         FROM student
         GROUP BY student_name;
  ```

- BIT_AND(*expr*) [*over_clause*]

  Returns the bitwise AND of all bits in *expr*.

  The result type depends on whether the function argument values are evaluated as binary strings or numbers:

  - Binary-string evaluation occurs when the argument values have a binary string type, and the argument is not a hexadecimal literal, bit literal, or NULL literal. Numeric evaluation occurs otherwise, with argument value conversion to unsigned 64-bit integers as necessary.
  - Binary-string evaluation produces a binary string of the same length as the argument values. If argument values have unequal lengths, an ER_INVALID_BITWISE_OPERANDS_SIZE error occurs. If the argument size exceeds 511 bytes, an ER_INVALID_BITWISE_AGGREGATE_OPERANDS_SIZE error occurs. Numeric evaluation produces an unsigned 64-bit integer.

  If there are no matching rows, BIT_AND() returns a neutral value (all bits set to 1) having the same length as the argument values.

`NULL` values do not affect the result unless all values are `NULL`. In that case, the result is a neutral value having the same length as the argument values.

For more information discussion about argument evaluation and result types, see the introductory discussion in Section 12.13, “Bit Functions and Operators”.

If `BIT_AND()` is invoked from within the `mysql` client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

As of MySQL 8.0.12, this function executes as a window function if *`over_clause`* is present. *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

- `BIT_OR(`*`expr`*`) [`*`over_clause`*`]`

  Returns the bitwise `OR` of all bits in *`expr`*.

  The result type depends on whether the function argument values are evaluated as binary strings or numbers:

  - Binary-string evaluation occurs when the argument values have a binary string type, and the argument is not a hexadecimal literal, bit literal, or `NULL` literal. Numeric evaluation occurs otherwise, with argument value conversion to unsigned 64-bit integers as necessary.
  - Binary-string evaluation produces a binary string of the same length as the argument values. If argument values have unequal lengths, an `ER_INVALID_BITWISE_OPERANDS_SIZE` error occurs. If the argument size exceeds 511 bytes, an `ER_INVALID_BITWISE_AGGREGATE_OPERANDS_SIZE` error occurs. Numeric evaluation produces an unsigned 64-bit integer.

  If there are no matching rows, `BIT_OR()` returns a neutral value (all bits set to 0) having the same length as the argument values.

  `NULL` values do not affect the result unless all values are `NULL`. In that case, the result is a neutral value having the same length as the argument values.

  For more information discussion about argument evaluation and result types, see the introductory discussion in Section 12.13, “Bit Functions and Operators”.

  If `BIT_OR()` is invoked from within the `mysql` client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

  As of MySQL 8.0.12, this function executes as a window function if *`over_clause`* is present. *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

- `BIT_XOR(`*`expr`*`) [`*`over_clause`*`]`

  Returns the bitwise `XOR` of all bits in *`expr`*.

  The result type depends on whether the function argument values are evaluated as binary strings or numbers:

  - Binary-string evaluation occurs when the argument values have a binary string type, and the argument is not a hexadecimal literal, bit literal, or `NULL` literal. Numeric evaluation occurs otherwise, with argument value conversion to unsigned 64-bit integers as necessary.
  - Binary-string evaluation produces a binary string of the same length as the argument values. If argument values have unequal lengths, an `ER_INVALID_BITWISE_OPERANDS_SIZE` error occurs. If the argument size exceeds 511 bytes,

an ER_INVALID_BITWISE_AGGREGATE_OPERANDS_SIZE error occurs. Numeric evaluation produces an unsigned 64-bit integer.

If there are no matching rows, BIT_XOR() returns a neutral value (all bits set to 0) having the same length as the argument values.

NULL values do not affect the result unless all values are NULL. In that case, the result is a neutral value having the same length as the argument values.

For more information discussion about argument evaluation and result types, see the introductory discussion in Section 12.13, “Bit Functions and Operators”.

If BIT_XOR() is invoked from within the mysql client, binary string results display using hexadecimal notation, depending on the value of the --binary-as-hex. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

As of MySQL 8.0.12, this function executes as a window function if *over_clause* is present. *over_clause* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

- COUNT(*expr*) [*over_clause*]

  Returns a count of the number of non-NULL values of *expr* in the rows retrieved by a SELECT statement. The result is a BIGINT value.

  If there are no matching rows, COUNT() returns 0. COUNT(NULL) returns 0.

  This function executes as a window function if *over_clause* is present. *over_clause* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

```
mysql> SELECT student.student_name,COUNT(*)
       FROM student,course
       WHERE student.student_id=course.student_id
       GROUP BY student_name;
```

  COUNT(*) is somewhat different in that it returns a count of the number of rows retrieved, whether or not they contain NULL values.

  For transactional storage engines such as InnoDB, storing an exact row count is problematic. Multiple transactions may be occurring at the same time, each of which may affect the count.

  InnoDB does not keep an internal count of rows in a table because concurrent transactions might “see” different numbers of rows at the same time. Consequently, SELECT COUNT(*) statements only count rows visible to the current transaction.

  As of MySQL 8.0.13, SELECT COUNT(*) FROM *tbl_name* query performance for InnoDB tables is optimized for single-threaded workloads if there are no extra clauses such as WHERE or GROUP BY.

  InnoDB processes SELECT COUNT(*) statements by traversing the smallest available secondary index unless an index or optimizer hint directs the optimizer to use a different index. If a secondary index is not present, InnoDB processes SELECT COUNT(*) statements by scanning the clustered index.

  Processing SELECT COUNT(*) statements takes some time if index records are not entirely in the buffer pool. For a faster count, create a counter table and let your application update it according to the inserts and deletes it does. However, this method may not scale well in situations

where thousands of concurrent transactions are initiating updates to the same counter table. If an approximate row count is sufficient, use `SHOW TABLE STATUS`.

`InnoDB` handles `SELECT COUNT(*)` and `SELECT COUNT(1)` operations in the same way. There is no performance difference.

For `MyISAM` tables, `COUNT(*)` is optimized to return very quickly if the `SELECT` retrieves from one table, no other columns are retrieved, and there is no `WHERE` clause. For example:

```
mysql> SELECT COUNT(*) FROM student;
```

This optimization only applies to `MyISAM` tables, because an exact row count is stored for this storage engine and can be accessed very quickly. `COUNT(1)` is only subject to the same optimization if the first column is defined as `NOT NULL`.

- `COUNT(DISTINCT expr,[expr...])`

  Returns a count of the number of rows with different non-`NULL` *`expr`* values.

  If there are no matching rows, `COUNT(DISTINCT)` returns `0`.

```
mysql> SELECT COUNT(DISTINCT results) FROM student;
```

  In MySQL, you can obtain the number of distinct expression combinations that do not contain `NULL` by giving a list of expressions. In standard SQL, you would have to do a concatenation of all expressions inside `COUNT(DISTINCT ...)`.

- `GROUP_CONCAT(expr)`

  This function returns a string result with the concatenated non-`NULL` values from a group. It returns `NULL` if there are no non-`NULL` values. The full syntax is as follows:

```
GROUP_CONCAT([DISTINCT] expr [,expr ...]
             [ORDER BY {unsigned_integer | col_name | expr}
                 [ASC | DESC] [,col_name ...]]
             [SEPARATOR str_val])
```

```
mysql> SELECT student_name,
         GROUP_CONCAT(test_score)
       FROM student
       GROUP BY student_name;
```

  Or:

```
mysql> SELECT student_name,
         GROUP_CONCAT(DISTINCT test_score
                      ORDER BY test_score DESC SEPARATOR ' ')
       FROM student
       GROUP BY student_name;
```

  In MySQL, you can get the concatenated values of expression combinations. To eliminate duplicate values, use the `DISTINCT` clause. To sort values in the result, use the `ORDER BY` clause. To sort in reverse order, add the `DESC` (descending) keyword to the name of the column you are sorting by in the `ORDER BY` clause. The default is ascending order; this may be specified explicitly using the `ASC` keyword. The default separator between values in a group is comma (`,`). To specify a separator explicitly, use `SEPARATOR` followed by the string literal value that should be inserted between group values. To eliminate the separator altogether, specify `SEPARATOR ''`.

  The result is truncated to the maximum length that is given by the `group_concat_max_len` system variable, which has a default value of 1024. The value can be set higher, although the effective maximum length of the return value is constrained by the value of `max_allowed_packet`. The

syntax to change the value of `group_concat_max_len` at runtime is as follows, where *`val`* is an unsigned integer:

```
SET [GLOBAL | SESSION] group_concat_max_len = val;
```

The return value is a nonbinary or binary string, depending on whether the arguments are nonbinary or binary strings. The result type is `TEXT` or `BLOB` unless `group_concat_max_len` is less than or equal to 512, in which case the result type is `VARCHAR` or `VARBINARY`.

If `GROUP_CONCAT()` is invoked from within the `mysql` client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

See also `CONCAT()` and `CONCAT_WS()`: Section 12.8, “String Functions and Operators”.

- `JSON_ARRAYAGG(`*`col_or_expr`*`) [`*`over_clause`*`]`

  Aggregates a result set as a single `JSON` array whose elements consist of the rows. The order of elements in this array is undefined. The function acts on a column or an expression that evaluates to a single value. Returns `NULL` if the result contains no rows, or in the event of an error. If *`col_or_expr`* is `NULL`, the function returns an array of JSON `[null]` elements.

  As of MySQL 8.0.14, this function executes as a window function if *`over_clause`* is present. *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

```
mysql> SELECT o_id, attribute, value FROM t3;
+------+-----------+-------+
| o_id | attribute | value |
+------+-----------+-------+
|    2 | color     | red   |
|    2 | fabric    | silk  |
|    3 | color     | green |
|    3 | shape     | square|
+------+-----------+-------+
4 rows in set (0.00 sec)

mysql> SELECT o_id, JSON_ARRAYAGG(attribute) AS attributes
    -> FROM t3 GROUP BY o_id;
+------+---------------------+
| o_id | attributes          |
+------+---------------------+
|    2 | ["color", "fabric"] |
|    3 | ["color", "shape"]  |
+------+---------------------+
2 rows in set (0.00 sec)
```

- `JSON_OBJECTAGG(`*`key, value`*`) [`*`over_clause`*`]`

  Takes two column names or expressions as arguments, the first of these being used as a key and the second as a value, and returns a JSON object containing key-value pairs. Returns `NULL` if the result contains no rows, or in the event of an error. An error occurs if any key name is `NULL` or the number of arguments is not equal to 2.

  As of MySQL 8.0.14, this function executes as a window function if *`over_clause`* is present. *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

```
mysql> SELECT o_id, attribute, value FROM t3;
+------+-----------+-------+
| o_id | attribute | value |
+------+-----------+-------+
|    2 | color     | red   |
|    2 | fabric    | silk  |
|    3 | color     | green |
|    3 | shape     | square|
+------+-----------+-------+
4 rows in set (0.00 sec)
```

```
mysql> SELECT o_id, JSON_OBJECTAGG(attribute, value)
    -> FROM t3 GROUP BY o_id;
+------+--------------------------------------+
| o_id | JSON_OBJECTAGG(attribute, value)     |
+------+--------------------------------------+
|    2 | {"color": "red", "fabric": "silk"}   |
|    3 | {"color": "green", "shape": "square"} |
+------+--------------------------------------+
2 rows in set (0.00 sec)
```

**Duplicate key handling.** When the result of this function is normalized, values having duplicate keys are discarded. In keeping with the MySQL `JSON` data type specification that does not permit duplicate keys, only the last value encountered is used with that key in the returned object (“last duplicate key wins”). This means that the result of using this function on columns from a `SELECT` can depend on the order in which the rows are returned, which is not guaranteed.

When used as a window function, if there are duplicate keys within a frame, only the last value for the key is present in the result. The value for the key from the last row in the frame is deterministic if the `ORDER BY` specification guarantees that the values have a specific order. If not, the resulting value of the key is nondeterministic.

Consider the following:

```
mysql> CREATE TABLE t(c VARCHAR(10), i INT);
Query OK, 0 rows affected (0.33 sec)

mysql> INSERT INTO t VALUES ('key', 3), ('key', 4), ('key', 5);
Query OK, 3 rows affected (0.10 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT c, i FROM t;
+------+------+
| c    | i    |
+------+------+
| key  |    3 |
| key  |    4 |
| key  |    5 |
+------+------+
3 rows in set (0.00 sec)

mysql> SELECT JSON_OBJECTAGG(c, i) FROM t;
+----------------------+
| JSON_OBJECTAGG(c, i) |
+----------------------+
| {"key": 5}           |
+----------------------+
1 row in set (0.00 sec)

mysql> DELETE FROM t;
Query OK, 3 rows affected (0.08 sec)

mysql> INSERT INTO t VALUES ('key', 3), ('key', 5), ('key', 4);
Query OK, 3 rows affected (0.06 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT c, i FROM t;
+------+------+
| c    | i    |
+------+------+
| key  |    3 |
| key  |    5 |
| key  |    4 |
+------+------+
3 rows in set (0.00 sec)

mysql> SELECT JSON_OBJECTAGG(c, i) FROM t;
+----------------------+
| JSON_OBJECTAGG(c, i) |
+----------------------+
```

```
| {"key": 4}           |
+----------------------+
1 row in set (0.00 sec)
```

The key chosen from the last query is nondeterministic. If the query does not use `GROUP BY` (which usually imposes its own ordering regardless) and you prefer a particular key ordering, you can invoke `JSON_OBJECTAGG()` as a window function by including an `OVER` clause with an `ORDER BY` specification to impose a particular order on frame rows. The following examples show what happens with and without `ORDER BY` for a few different frame specifications.

Without `ORDER BY`, the frame is the entire partition:

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER () AS json_object FROM t;
+-------------+
| json_object |
+-------------+
| {"key": 4}  |
| {"key": 4}  |
| {"key": 4}  |
+-------------+
```

With `ORDER BY`, where the frame is the default of `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` (in both ascending and descending order):

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i) AS json_object FROM t;
+-------------+
| json_object |
+-------------+
| {"key": 3}  |
| {"key": 4}  |
| {"key": 5}  |
+-------------+
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i DESC) AS json_object FROM t;
+-------------+
| json_object |
+-------------+
| {"key": 5}  |
| {"key": 4}  |
| {"key": 3}  |
+-------------+
```

With `ORDER BY` and an explicit frame of the entire partition:

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i
            ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
        AS json_object
       FROM t;
+-------------+
| json_object |
+-------------+
| {"key": 5}  |
| {"key": 5}  |
| {"key": 5}  |
+-------------+
```

To return a particular key value (such as the smallest or largest), include a `LIMIT` clause in the appropriate query. For example:

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i) AS json_object FROM t LIMIT 1;
+-------------+
| json_object |
+-------------+
| {"key": 3}  |
+-------------+
```

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i DESC) AS json_object FROM t LIMIT 1;
+-------------+
| json_object |
+-------------+
| {"key": 5}  |
+-------------+
```

See Normalization, Merging, and Autowrapping of JSON Values, for additional information and examples.

- `MAX([DISTINCT] expr) [over_clause]`

  Returns the maximum value of *`expr`*. `MAX()` may take a string argument; in such cases, it returns the maximum string value. See Section 8.3.1, “How MySQL Uses Indexes”. The `DISTINCT` keyword can be used to find the maximum of the distinct values of *`expr`*, however, this produces the same result as omitting `DISTINCT`.

  If there are no matching rows, or if *`expr`* is `NULL`, `MAX()` returns `NULL`.

  This function executes as a window function if *`over_clause`* is present. *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”; it cannot be used with `DISTINCT`.

```
mysql> SELECT student_name, MIN(test_score), MAX(test_score)
       FROM student
       GROUP BY student_name;
```

  For `MAX()`, MySQL currently compares `ENUM` and `SET` columns by their string value rather than by the string's relative position in the set. This differs from how `ORDER BY` compares them.

- `MIN([DISTINCT] expr) [over_clause]`

  Returns the minimum value of *`expr`*. `MIN()` may take a string argument; in such cases, it returns the minimum string value. See Section 8.3.1, “How MySQL Uses Indexes”. The `DISTINCT` keyword can be used to find the minimum of the distinct values of *`expr`*, however, this produces the same result as omitting `DISTINCT`.

  If there are no matching rows, or if *`expr`* is `NULL`, `MIN()` returns `NULL`.

  This function executes as a window function if *`over_clause`* is present. *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”; it cannot be used with `DISTINCT`.

```
mysql> SELECT student_name, MIN(test_score), MAX(test_score)
       FROM student
       GROUP BY student_name;
```

  For `MIN()`, MySQL currently compares `ENUM` and `SET` columns by their string value rather than by the string's relative position in the set. This differs from how `ORDER BY` compares them.

- `STD(expr) [over_clause]`

  Returns the population standard deviation of *`expr`*. `STD()` is a synonym for the standard SQL function `STDDEV_POP()`, provided as a MySQL extension.

  If there are no matching rows, or if *`expr`* is `NULL`, `STD()` returns `NULL`.

  This function executes as a window function if *`over_clause`* is present. *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

- `STDDEV(expr) [over_clause]`

  Returns the population standard deviation of `expr`. `STDDEV()` is a synonym for the standard SQL function `STDDEV_POP()`, provided for compatibility with Oracle.

  If there are no matching rows, or if `expr` is `NULL`, `STDDEV()` returns `NULL`.

  This function executes as a window function if `over_clause` is present. `over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

- `STDDEV_POP(expr) [over_clause]`

  Returns the population standard deviation of `expr` (the square root of `VAR_POP()`). You can also use `STD()` or `STDDEV()`, which are equivalent but not standard SQL.

  If there are no matching rows, or if `expr` is `NULL`, `STDDEV_POP()` returns `NULL`.

  This function executes as a window function if `over_clause` is present. `over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

- `STDDEV_SAMP(expr) [over_clause]`

  Returns the sample standard deviation of `expr` (the square root of `VAR_SAMP()`.

  If there are no matching rows, or if `expr` is `NULL`, `STDDEV_SAMP()` returns `NULL`.

  This function executes as a window function if `over_clause` is present. `over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

- `SUM([DISTINCT] expr) [over_clause]`

  Returns the sum of `expr`. If the return set has no rows, `SUM()` returns `NULL`. The `DISTINCT` keyword can be used to sum only the distinct values of `expr`.

  If there are no matching rows, or if `expr` is `NULL`, `SUM()` returns `NULL`.

  This function executes as a window function if `over_clause` is present. `over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”; it cannot be used with `DISTINCT`.

- `VAR_POP(expr) [over_clause]`

  Returns the population standard variance of `expr`. It considers rows as the whole population, not as a sample, so it has the number of rows as the denominator. You can also use `VARIANCE()`, which is equivalent but is not standard SQL.

  If there are no matching rows, or if `expr` is `NULL`, `VAR_POP()` returns `NULL`.

  This function executes as a window function if `over_clause` is present. `over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

- `VAR_SAMP(expr) [over_clause]`

  Returns the sample variance of `expr`. That is, the denominator is the number of rows minus one.

  If there are no matching rows, or if `expr` is `NULL`, `VAR_SAMP()` returns `NULL`.

  This function executes as a window function if `over_clause` is present. `over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

- `VARIANCE(expr) [over_clause]`

  Returns the population standard variance of *`expr`*. `VARIANCE()` is a synonym for the standard SQL function `VAR_POP()`, provided as a MySQL extension.

  If there are no matching rows, or if *`expr`* is `NULL`, `VARIANCE()` returns `NULL`.

  This function executes as a window function if *`over_clause`* is present. *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

## 12.20.2 GROUP BY Modifiers

The `GROUP BY` clause permits a `WITH ROLLUP` modifier that causes summary output to include extra rows that represent higher-level (that is, super-aggregate) summary operations. `ROLLUP` thus enables you to answer questions at multiple levels of analysis with a single query. For example, `ROLLUP` can be used to provide support for OLAP (Online Analytical Processing) operations.

Suppose that a `sales` table has `year`, `country`, `product`, and `profit` columns for recording sales profitability:

```
CREATE TABLE sales
(
    year    INT,
    country VARCHAR(20),
    product VARCHAR(32),
    profit  INT
);
```

To summarize table contents per year, use a simple `GROUP BY` like this:

```
mysql> SELECT year, SUM(profit) AS profit
       FROM sales
       GROUP BY year;
+------+--------+
| year | profit |
+------+--------+
| 2000 |   4525 |
| 2001 |   3010 |
+------+--------+
```

The output shows the total (aggregate) profit for each year. To also determine the total profit summed over all years, you must add up the individual values yourself or run an additional query. Or you can use `ROLLUP`, which provides both levels of analysis with a single query. Adding a `WITH ROLLUP` modifier to the `GROUP BY` clause causes the query to produce another (super-aggregate) row that shows the grand total over all year values:

```
mysql> SELECT year, SUM(profit) AS profit
       FROM sales
       GROUP BY year WITH ROLLUP;
+------+--------+
| year | profit |
+------+--------+
| 2000 |   4525 |
| 2001 |   3010 |
| NULL |   7535 |
+------+--------+
```

The `NULL` value in the `year` column identifies the grand total super-aggregate line.

`ROLLUP` has a more complex effect when there are multiple `GROUP BY` columns. In this case, each time there is a change in value in any but the last grouping column, the query produces an extra super-aggregate summary row.

For example, without `ROLLUP`, a summary of the `sales` table based on `year`, `country`, and `product` might look like this, where the output indicates summary values only at the year/country/product level of analysis:

```
mysql> SELECT year, country, product, SUM(profit) AS profit
       FROM sales
       GROUP BY year, country, product;
+------+---------+------------+--------+
| year | country | product    | profit |
+------+---------+------------+--------+
| 2000 | Finland | Computer   |   1500 |
| 2000 | Finland | Phone      |    100 |
| 2000 | India   | Calculator |    150 |
| 2000 | India   | Computer   |   1200 |
| 2000 | USA     | Calculator |     75 |
| 2000 | USA     | Computer   |   1500 |
| 2001 | Finland | Phone      |     10 |
| 2001 | USA     | Calculator |     50 |
| 2001 | USA     | Computer   |   2700 |
| 2001 | USA     | TV         |    250 |
+------+---------+------------+--------+
```

With `ROLLUP` added, the query produces several extra rows:

```
mysql> SELECT year, country, product, SUM(profit) AS profit
       FROM sales
       GROUP BY year, country, product WITH ROLLUP;
+------+---------+------------+--------+
| year | country | product    | profit |
+------+---------+------------+--------+
| 2000 | Finland | Computer   |   1500 |
| 2000 | Finland | Phone      |    100 |
| 2000 | Finland | NULL       |   1600 |
| 2000 | India   | Calculator |    150 |
| 2000 | India   | Computer   |   1200 |
| 2000 | India   | NULL       |   1350 |
| 2000 | USA     | Calculator |     75 |
| 2000 | USA     | Computer   |   1500 |
| 2000 | USA     | NULL       |   1575 |
| 2000 | NULL    | NULL       |   4525 |
| 2001 | Finland | Phone      |     10 |
| 2001 | Finland | NULL       |     10 |
| 2001 | USA     | Calculator |     50 |
| 2001 | USA     | Computer   |   2700 |
| 2001 | USA     | TV         |    250 |
| 2001 | USA     | NULL       |   3000 |
| 2001 | NULL    | NULL       |   3010 |
| NULL | NULL    | NULL       |   7535 |
+------+---------+------------+--------+
```

Now the output includes summary information at four levels of analysis, not just one:

- Following each set of product rows for a given year and country, an extra super-aggregate summary row appears showing the total for all products. These rows have the `product` column set to `NULL`.
- Following each set of rows for a given year, an extra super-aggregate summary row appears showing the total for all countries and products. These rows have the `country` and `products` columns set to `NULL`.
- Finally, following all other rows, an extra super-aggregate summary row appears showing the grand total for all years, countries, and products. This row has the `year`, `country`, and `products` columns set to `NULL`.

The `NULL` indicators in each super-aggregate row are produced when the row is sent to the client. The server looks at the columns named in the `GROUP BY` clause following the leftmost one that has changed value. For any column in the result set with a name that matches any of those names, its value is set to `NULL`. (If you specify grouping columns by column position, the server identifies which columns to set to `NULL` by position.)

Because the `NULL` values in the super-aggregate rows are placed into the result set at such a late stage in query processing, you can test them as `NULL` values only in the select list or `HAVING` clause. You cannot test them as `NULL` values in join conditions or the `WHERE` clause to determine which rows

to select. For example, you cannot add `WHERE product IS NULL` to the query to eliminate from the output all but the super-aggregate rows.

The `NULL` values do appear as `NULL` on the client side and can be tested as such using any MySQL client programming interface. However, at this point, you cannot distinguish whether a `NULL` represents a regular grouped value or a super-aggregate value. To test the distinction, use the `GROUPING()` function, described later.

Previously, MySQL did not allow the use of `DISTINCT` or `ORDER BY` in a query having a `WITH ROLLUP` option. This restriction is lifted in MySQL 8.0.12 and later. (Bug #87450, Bug #86311, Bug #26640100, Bug #26073513)

For `GROUP BY ... WITH ROLLUP` queries, to test whether `NULL` values in the result represent super-aggregate values, the `GROUPING()` function is available for use in the select list, `HAVING` clause, and (as of MySQL 8.0.12) `ORDER BY` clause. For example, `GROUPING(year)` returns 1 when `NULL` in the `year` column occurs in a super-aggregate row, and 0 otherwise. Similarly, `GROUPING(country)` and `GROUPING(product)` return 1 for super-aggregate `NULL` values in the `country` and `product` columns, respectively:

```
mysql> SELECT
         year, country, product, SUM(profit) AS profit,
         GROUPING(year) AS grp_year,
         GROUPING(country) AS grp_country,
         GROUPING(product) AS grp_product
       FROM sales
       GROUP BY year, country, product WITH ROLLUP;
+------+---------+------------+--------+----------+-------------+-------------+
| year | country | product    | profit | grp_year | grp_country | grp_product |
+------+---------+------------+--------+----------+-------------+-------------+
| 2000 | Finland | Computer   |   1500 |        0 |           0 |           0 |
| 2000 | Finland | Phone      |    100 |        0 |           0 |           0 |
| 2000 | Finland | NULL       |   1600 |        0 |           0 |           1 |
| 2000 | India   | Calculator |    150 |        0 |           0 |           0 |
| 2000 | India   | Computer   |   1200 |        0 |           0 |           0 |
| 2000 | India   | NULL       |   1350 |        0 |           0 |           1 |
| 2000 | USA     | Calculator |     75 |        0 |           0 |           0 |
| 2000 | USA     | Computer   |   1500 |        0 |           0 |           0 |
| 2000 | USA     | NULL       |   1575 |        0 |           0 |           1 |
| 2000 | NULL    | NULL       |   4525 |        0 |           1 |           1 |
| 2001 | Finland | Phone      |     10 |        0 |           0 |           0 |
| 2001 | Finland | NULL       |     10 |        0 |           0 |           1 |
| 2001 | USA     | Calculator |     50 |        0 |           0 |           0 |
| 2001 | USA     | Computer   |   2700 |        0 |           0 |           0 |
| 2001 | USA     | TV         |    250 |        0 |           0 |           0 |
| 2001 | USA     | NULL       |   3000 |        0 |           0 |           1 |
| 2001 | NULL    | NULL       |   3010 |        0 |           1 |           1 |
| NULL | NULL    | NULL       |   7535 |        1 |           1 |           1 |
+------+---------+------------+--------+----------+-------------+-------------+
```

Instead of displaying the `GROUPING()` results directly, you can use `GROUPING()` to substitute labels for super-aggregate `NULL` values:

```
mysql> SELECT
         IF(GROUPING(year), 'All years', year) AS year,
         IF(GROUPING(country), 'All countries', country) AS country,
         IF(GROUPING(product), 'All products', product) AS product,
         SUM(profit) AS profit
       FROM sales
       GROUP BY year, country, product WITH ROLLUP;
+-----------+---------------+--------------+--------+
| year      | country       | product      | profit |
+-----------+---------------+--------------+--------+
| 2000      | Finland       | Computer     |   1500 |
| 2000      | Finland       | Phone        |    100 |
| 2000      | Finland       | All products |   1600 |
| 2000      | India         | Calculator   |    150 |
| 2000      | India         | Computer     |   1200 |
| 2000      | India         | All products |   1350 |
```

```
| 2000      | USA           | Calculator   |     75 |
| 2000      | USA           | Computer     |   1500 |
| 2000      | USA           | All products |   1575 |
| 2000      | All countries | All products |   4525 |
| 2001      | Finland       | Phone        |     10 |
| 2001      | Finland       | All products |     10 |
| 2001      | USA           | Calculator   |     50 |
| 2001      | USA           | Computer     |   2700 |
| 2001      | USA           | TV           |    250 |
| 2001      | USA           | All products |   3000 |
| 2001      | All countries | All products |   3010 |
| All years | All countries | All products |   7535 |
+-----------+---------------+--------------+--------+
```

With multiple expression arguments, `GROUPING()` returns a result representing a bitmask the combines the results for each expression, with the lowest-order bit corresponding to the result for the rightmost expression. For example, `GROUPING(year, country, product)` is evaluated like this:

```
  result for GROUPING(product)
+ result for GROUPING(country) << 1
+ result for GROUPING(year) << 2
```

The result of such a `GROUPING()` is nonzero if any of the expressions represents a super-aggregate `NULL`, so you can return only the super-aggregate rows and filter out the regular grouped rows like this:

```
mysql> SELECT year, country, product, SUM(profit) AS profit
       FROM sales
       GROUP BY year, country, product WITH ROLLUP
       HAVING GROUPING(year, country, product) <> 0;
+------+---------+---------+--------+
| year | country | product | profit |
+------+---------+---------+--------+
| 2000 | Finland | NULL    |   1600 |
| 2000 | India   | NULL    |   1350 |
| 2000 | USA     | NULL    |   1575 |
| 2000 | NULL    | NULL    |   4525 |
| 2001 | Finland | NULL    |     10 |
| 2001 | USA     | NULL    |   3000 |
| 2001 | NULL    | NULL    |   3010 |
| NULL | NULL    | NULL    |   7535 |
+------+---------+---------+--------+
```

The `sales` table contains no `NULL` values, so all `NULL` values in a `ROLLUP` result represent super-aggregate values. When the data set contains `NULL` values, `ROLLUP` summaries may contain `NULL` values not only in super-aggregate rows, but also in regular grouped rows. `GROUPING()` enables these to be distinguished. Suppose that table `t1` contains a simple data set with two grouping factors for a set of quantity values, where `NULL` indicates something like “other” or “unknown”:

```
mysql> SELECT * FROM t1;
+------+-------+----------+
| name | size  | quantity |
+------+-------+----------+
| ball | small |       10 |
| ball | large |       20 |
| ball | NULL  |        5 |
| hoop | small |       15 |
| hoop | large |        5 |
| hoop | NULL  |        3 |
+------+-------+----------+
```

A simple `ROLLUP` operation produces these results, in which it is not so easy to distinguish `NULL` values in super-aggregate rows from `NULL` values in regular grouped rows:

```
mysql> SELECT name, size, SUM(quantity) AS quantity
       FROM t1
       GROUP BY name, size WITH ROLLUP;
+------+-------+----------+
| name | size  | quantity |
+------+-------+----------+
| ball | NULL  |        5 |
```

```
| ball | large |       20 |
| ball | small |       10 |
| ball | NULL  |       35 |
| hoop | NULL  |        3 |
| hoop | large |        5 |
| hoop | small |       15 |
| hoop | NULL  |       23 |
| NULL | NULL  |       58 |
+------+-------+----------+
```

Using `GROUPING()` to substitute labels for the super-aggregate `NULL` values makes the result easier to interpret:

```
mysql> SELECT
         IF(GROUPING(name) = 1, 'All items', name) AS name,
         IF(GROUPING(size) = 1, 'All sizes', size) AS size,
         SUM(quantity) AS quantity
       FROM t1
       GROUP BY name, size WITH ROLLUP;
+-----------+-----------+----------+
| name      | size      | quantity |
+-----------+-----------+----------+
| ball      | NULL      |        5 |
| ball      | large     |       20 |
| ball      | small     |       10 |
| ball      | All sizes |       35 |
| hoop      | NULL      |        3 |
| hoop      | large     |        5 |
| hoop      | small     |       15 |
| hoop      | All sizes |       23 |
| All items | All sizes |       58 |
+-----------+-----------+----------+
```

## Other Considerations When using ROLLUP

The following discussion lists some behaviors specific to the MySQL implementation of `ROLLUP`.

Prior to MySQL 8.0.12, when you use `ROLLUP`, you cannot also use an `ORDER BY` clause to sort the results. In other words, `ROLLUP` and `ORDER BY` were mutually exclusive in MySQL. However, you still have some control over sort order. To work around the restriction that prevents using `ROLLUP` with `ORDER BY` and achieve a specific sort order of grouped results, generate the grouped result set as a derived table and apply `ORDER BY` to it. For example:

```
mysql> SELECT * FROM
         (SELECT year, SUM(profit) AS profit
         FROM sales GROUP BY year WITH ROLLUP) AS dt
       ORDER BY year DESC;
+------+--------+
| year | profit |
+------+--------+
| 2001 |   3010 |
| 2000 |   4525 |
| NULL |   7535 |
+------+--------+
```

As of MySQL 8.0.12, `ORDER BY` and `ROLLUP` can be used together, which enables the use of `ORDER BY` and `GROUPING()` to achieve a specific sort order of grouped results. For example:

```
mysql> SELECT year, SUM(profit) AS profit
       FROM sales
       GROUP BY year WITH ROLLUP
       ORDER BY GROUPING(year) DESC;
+------+--------+
| year | profit |
+------+--------+
| NULL |   7535 |
| 2000 |   4525 |
| 2001 |   3010 |
+------+--------+
```

In both cases, the super-aggregate summary rows sort with the rows from which they are calculated, and their placement depends on sort order (at the end for ascending sort, at the beginning for descending sort).

`LIMIT` can be used to restrict the number of rows returned to the client. `LIMIT` is applied after `ROLLUP`, so the limit applies against the extra rows added by `ROLLUP`. For example:

```
mysql> SELECT year, country, product, SUM(profit) AS profit
       FROM sales
       GROUP BY year, country, product WITH ROLLUP
       LIMIT 5;
+------+---------+------------+--------+
| year | country | product    | profit |
+------+---------+------------+--------+
| 2000 | Finland | Computer   |   1500 |
| 2000 | Finland | Phone      |    100 |
| 2000 | Finland | NULL       |   1600 |
| 2000 | India   | Calculator |    150 |
| 2000 | India   | Computer   |   1200 |
+------+---------+------------+--------+
```

Using `LIMIT` with `ROLLUP` may produce results that are more difficult to interpret, because there is less context for understanding the super-aggregate rows.

A MySQL extension permits a column that does not appear in the `GROUP BY` list to be named in the select list. (For information about nonaggregated columns and `GROUP BY`, see Section 12.20.3, “MySQL Handling of GROUP BY”.) In this case, the server is free to choose any value from this nonaggregated column in summary rows, and this includes the extra rows added by `WITH ROLLUP`. For example, in the following query, `country` is a nonaggregated column that does not appear in the `GROUP BY` list and values chosen for this column are nondeterministic:

```
mysql> SELECT year, country, SUM(profit) AS profit
       FROM sales
       GROUP BY year WITH ROLLUP;
+------+---------+--------+
| year | country | profit |
+------+---------+--------+
| 2000 | India   |   4525 |
| 2001 | USA     |   3010 |
| NULL | USA     |   7535 |
+------+---------+--------+
```

This behavior is permitted when the `ONLY_FULL_GROUP_BY` SQL mode is not enabled. If that mode is enabled, the server rejects the query as illegal because `country` is not listed in the `GROUP BY` clause. With `ONLY_FULL_GROUP_BY` enabled, you can still execute the query by using the `ANY_VALUE()` function for nondeterministic-value columns:

```
mysql> SELECT year, ANY_VALUE(country) AS country, SUM(profit) AS profit
       FROM sales
       GROUP BY year WITH ROLLUP;
+------+---------+--------+
| year | country | profit |
+------+---------+--------+
| 2000 | India   |   4525 |
| 2001 | USA     |   3010 |
| NULL | USA     |   7535 |
+------+---------+--------+
```

In MySQL 8.0.28 and later, a rollup column cannot be used as an argument to `MATCH()` (and is rejected with an error) except when called in a `WHERE` clause. See Section 12.10, “Full-Text Search Functions”, for more information.

## 12.20.3 MySQL Handling of GROUP BY

SQL-92 and earlier does not permit queries for which the select list, `HAVING` condition, or `ORDER BY` list refer to nonaggregated columns that are not named in the `GROUP BY` clause. For example, this

query is illegal in standard SQL-92 because the nonaggregated `name` column in the select list does not appear in the `GROUP BY`:

```
SELECT o.custid, c.name, MAX(o.payment)
  FROM orders AS o, customers AS c
  WHERE o.custid = c.custid
  GROUP BY o.custid;
```

For the query to be legal in SQL-92, the `name` column must be omitted from the select list or named in the `GROUP BY` clause.

SQL:1999 and later permits such nonaggregates per optional feature T301 if they are functionally dependent on `GROUP BY` columns: If such a relationship exists between `name` and `custid`, the query is legal. This would be the case, for example, were `custid` a primary key of `customers`.

MySQL implements detection of functional dependence. If the `ONLY_FULL_GROUP_BY` SQL mode is enabled (which it is by default), MySQL rejects queries for which the select list, `HAVING` condition, or `ORDER BY` list refer to nonaggregated columns that are neither named in the `GROUP BY` clause nor are functionally dependent on them.

MySQL also permits a nonaggregate column not named in a `GROUP BY` clause when SQL `ONLY_FULL_GROUP_BY` mode is enabled, provided that this column is limited to a single value, as shown in the following example:

```
mysql> CREATE TABLE mytable (
    ->    id INT UNSIGNED NOT NULL PRIMARY KEY,
    ->    a VARCHAR(10),
    ->    b INT
    -> );

mysql> INSERT INTO mytable
    -> VALUES (1, 'abc', 1000),
    ->        (2, 'abc', 2000),
    ->        (3, 'def', 4000);

mysql> SET SESSION sql_mode = sys.list_add(@@session.sql_mode, 'ONLY_FULL_GROUP_BY');

mysql> SELECT a, SUM(b) FROM mytable WHERE a = 'abc';
+------+--------+
| a    | SUM(b) |
+------+--------+
| abc  |   3000 |
+------+--------+
```

It is also possible to have more than one nonaggregate column in the `SELECT` list when employing `ONLY_FULL_GROUP_BY`. In this case, every such column must be limited to a single value in the `WHERE` clause, and all such limiting conditions must be joined by logical `AND`, as shown here:

```
mysql> DROP TABLE IF EXISTS mytable;

mysql> CREATE TABLE mytable (
    ->    id INT UNSIGNED NOT NULL PRIMARY KEY,
    ->    a VARCHAR(10),
    ->    b VARCHAR(10),
    ->    c INT
    -> );

mysql> INSERT INTO mytable
    -> VALUES (1, 'abc', 'qrs', 1000),
    ->        (2, 'abc', 'tuv', 2000),
    ->        (3, 'def', 'qrs', 4000),
    ->        (4, 'def', 'tuv', 8000),
    ->        (5, 'abc', 'qrs', 16000),
    ->        (6, 'def', 'tuv', 32000);

mysql> SELECT @@session.sql_mode;
+-------------------------------------------------------------+
| @@session.sql_mode                                          |
+-------------------------------------------------------------+
```

```
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION |
+---------------------------------------------------------------+

mysql> SELECT a, b, SUM(c) FROM mytable
    ->     WHERE a = 'abc' AND b = 'qrs';
+------+------+--------+
| a    | b    | SUM(c) |
+------+------+--------+
| abc  | qrs  |  17000 |
+------+------+--------+
```

If `ONLY_FULL_GROUP_BY` is disabled, a MySQL extension to the standard SQL use of `GROUP BY` permits the select list, `HAVING` condition, or `ORDER BY` list to refer to nonaggregated columns even if the columns are not functionally dependent on `GROUP BY` columns. This causes MySQL to accept the preceding query. In this case, the server is free to choose any value from each group, so unless they are the same, the values chosen are nondeterministic, which is probably not what you want. Furthermore, the selection of values from each group cannot be influenced by adding an `ORDER BY` clause. Result set sorting occurs after values have been chosen, and `ORDER BY` does not affect which value within each group the server chooses. Disabling `ONLY_FULL_GROUP_BY` is useful primarily when you know that, due to some property of the data, all values in each nonaggregated column not named in the `GROUP BY` are the same for each group.

You can achieve the same effect without disabling `ONLY_FULL_GROUP_BY` by using `ANY_VALUE()` to refer to the nonaggregated column.

The following discussion demonstrates functional dependence, the error message MySQL produces when functional dependence is absent, and ways of causing MySQL to accept a query in the absence of functional dependence.

This query might be invalid with `ONLY_FULL_GROUP_BY` enabled because the nonaggregated `address` column in the select list is not named in the `GROUP BY` clause:

```
SELECT name, address, MAX(age) FROM t GROUP BY name;
```

The query is valid if `name` is a primary key of `t` or is a unique `NOT NULL` column. In such cases, MySQL recognizes that the selected column is functionally dependent on a grouping column. For example, if `name` is a primary key, its value determines the value of `address` because each group has only one value of the primary key and thus only one row. As a result, there is no randomness in the choice of `address` value in a group and no need to reject the query.

The query is invalid if `name` is not a primary key of `t` or a unique `NOT NULL` column. In this case, no functional dependency can be inferred and an error occurs:

```
mysql> SELECT name, address, MAX(age) FROM t GROUP BY name;
ERROR 1055 (42000): Expression #2 of SELECT list is not in GROUP
BY clause and contains nonaggregated column 'mydb.t.address' which
is not functionally dependent on columns in GROUP BY clause; this
is incompatible with sql_mode=only_full_group_by
```

If you know that, *for a given data set,* each `name` value in fact uniquely determines the `address` value, `address` is effectively functionally dependent on `name`. To tell MySQL to accept the query, you can use the `ANY_VALUE()` function:

```
SELECT name, ANY_VALUE(address), MAX(age) FROM t GROUP BY name;
```

Alternatively, disable `ONLY_FULL_GROUP_BY`.

The preceding example is quite simple, however. In particular, it is unlikely you would group on a single primary key column because every group would contain only one row. For additional examples demonstrating functional dependence in more complex queries, see Section 12.20.4, “Detection of Functional Dependence”.

If a query has aggregate functions and no `GROUP BY` clause, it cannot have nonaggregated columns in the select list, `HAVING` condition, or `ORDER BY` list with `ONLY_FULL_GROUP_BY` enabled:

```
mysql> SELECT name, MAX(age) FROM t;
ERROR 1140 (42000): In aggregated query without GROUP BY, expression
#1 of SELECT list contains nonaggregated column 'mydb.t.name'; this
is incompatible with sql_mode=only_full_group_by
```

Without `GROUP BY`, there is a single group and it is nondeterministic which `name` value to choose for the group. Here, too, `ANY_VALUE()` can be used, if it is immaterial which `name` value MySQL chooses:

```
SELECT ANY_VALUE(name), MAX(age) FROM t;
```

`ONLY_FULL_GROUP_BY` also affects handling of queries that use `DISTINCT` and `ORDER BY`. Consider the case of a table `t` with three columns `c1`, `c2`, and `c3` that contains these rows:

```
c1 c2 c3
1  2  A
3  4  B
1  2  C
```

Suppose that we execute the following query, expecting the results to be ordered by `c3`:

```
SELECT DISTINCT c1, c2 FROM t ORDER BY c3;
```

To order the result, duplicates must be eliminated first. But to do so, should we keep the first row or the third? This arbitrary choice influences the retained value of `c3`, which in turn influences ordering and makes it arbitrary as well. To prevent this problem, a query that has `DISTINCT` and `ORDER BY` is rejected as invalid if any `ORDER BY` expression does not satisfy at least one of these conditions:

- The expression is equal to one in the select list
- All columns referenced by the expression and belonging to the query's selected tables are elements of the select list

Another MySQL extension to standard SQL permits references in the `HAVING` clause to aliased expressions in the select list. For example, the following query returns `name` values that occur only once in table `orders`:

```
SELECT name, COUNT(name) FROM orders
  GROUP BY name
  HAVING COUNT(name) = 1;
```

The MySQL extension permits the use of an alias in the `HAVING` clause for the aggregated column:

```
SELECT name, COUNT(name) AS c FROM orders
  GROUP BY name
  HAVING c = 1;
```

Standard SQL permits only column expressions in `GROUP BY` clauses, so a statement such as this is invalid because `FLOOR(value/100)` is a noncolumn expression:

```
SELECT id, FLOOR(value/100)
  FROM tbl_name
  GROUP BY id, FLOOR(value/100);
```

MySQL extends standard SQL to permit noncolumn expressions in `GROUP BY` clauses and considers the preceding statement valid.

Standard SQL also does not permit aliases in `GROUP BY` clauses. MySQL extends standard SQL to permit aliases, so another way to write the query is as follows:

```
SELECT id, FLOOR(value/100) AS val
  FROM tbl_name
  GROUP BY id, val;
```

The alias `val` is considered a column expression in the `GROUP BY` clause.

In the presence of a noncolumn expression in the `GROUP BY` clause, MySQL recognizes equality between that expression and expressions in the select list. This means that

with `ONLY_FULL_GROUP_BY` SQL mode enabled, the query containing `GROUP BY id, FLOOR(value/100)` is valid because that same `FLOOR()` expression occurs in the select list. However, MySQL does not try to recognize functional dependence on `GROUP BY` noncolumn expressions, so the following query is invalid with `ONLY_FULL_GROUP_BY` enabled, even though the third selected expression is a simple formula of the `id` column and the `FLOOR()` expression in the `GROUP BY` clause:

```
SELECT id, FLOOR(value/100), id+FLOOR(value/100)
  FROM tbl_name
  GROUP BY id, FLOOR(value/100);
```

A workaround is to use a derived table:

```
SELECT id, F, id+F
  FROM
    (SELECT id, FLOOR(value/100) AS F
     FROM tbl_name
     GROUP BY id, FLOOR(value/100)) AS dt;
```

## 12.20.4 Detection of Functional Dependence

The following discussion provides several examples of the ways in which MySQL detects functional dependencies. The examples use this notation:

```
{X} -> {Y}
```

Understand this as "*X* uniquely determines *Y*," which also means that *Y* is functionally dependent on *X*.

The examples use the `world` database, which can be downloaded from https://dev.mysql.com/doc/index-other.html. You can find details on how to install the database on the same page.

- Functional Dependencies Derived from Keys
- Functional Dependencies Derived from Multiple-Column Keys and from Equalities
- Functional Dependency Special Cases
- Functional Dependencies and Views
- Combinations of Functional Dependencies

### Functional Dependencies Derived from Keys

The following query selects, for each country, a count of spoken languages:

```
SELECT co.Name, COUNT(*)
FROM countrylanguage cl, country co
WHERE cl.CountryCode = co.Code
GROUP BY co.Code;
```

`co.Code` is a primary key of `co`, so all columns of `co` are functionally dependent on it, as expressed using this notation:

```
{co.Code} -> {co.*}
```

Thus, `co.name` is functionally dependent on `GROUP BY` columns and the query is valid.

A `UNIQUE` index over a `NOT NULL` column could be used instead of a primary key and the same functional dependence would apply. (This is not true for a `UNIQUE` index that permits `NULL` values because it permits multiple `NULL` values and in that case uniqueness is lost.)

### Functional Dependencies Derived from Multiple-Column Keys and from Equalities

This query selects, for each country, a list of all spoken languages and how many people speak them:

```
SELECT co.Name, cl.Language,
cl.Percentage * co.Population / 100.0 AS SpokenBy
```

```
FROM countrylanguage cl, country co
WHERE cl.CountryCode = co.Code
GROUP BY cl.CountryCode, cl.Language;
```

The pair (`cl.CountryCode`, `cl.Language`) is a two-column composite primary key of `cl`, so that column pair uniquely determines all columns of `cl`:

```
{cl.CountryCode, cl.Language} -> {cl.*}
```

Moreover, because of the equality in the `WHERE` clause:

```
{cl.CountryCode} -> {co.Code}
```

And, because `co.Code` is primary key of `co`:

```
{co.Code} -> {co.*}
```

“Uniquely determines” relationships are transitive, therefore:

```
{cl.CountryCode, cl.Language} -> {cl.*,co.*}
```

As a result, the query is valid.

As with the previous example, a `UNIQUE` key over `NOT NULL` columns could be used instead of a primary key.

An `INNER JOIN` condition can be used instead of `WHERE`. The same functional dependencies apply:

```
SELECT co.Name, cl.Language,
cl.Percentage * co.Population/100.0 AS SpokenBy
FROM countrylanguage cl INNER JOIN country co
ON cl.CountryCode = co.Code
GROUP BY cl.CountryCode, cl.Language;
```

## Functional Dependency Special Cases

Whereas an equality test in a `WHERE` condition or `INNER JOIN` condition is symmetric, an equality test in an outer join condition is not, because tables play different roles.

Assume that referential integrity has been accidentally broken and there exists a row of `countrylanguage` without a corresponding row in `country`. Consider the same query as in the previous example, but with a `LEFT JOIN`:

```
SELECT co.Name, cl.Language,
cl.Percentage * co.Population/100.0 AS SpokenBy
FROM countrylanguage cl LEFT JOIN country co
ON cl.CountryCode = co.Code
GROUP BY cl.CountryCode, cl.Language;
```

For a given value of `cl.CountryCode`, the value of `co.Code` in the join result is either found in a matching row (determined by `cl.CountryCode`) or is `NULL`-complemented if there is no match (also determined by `cl.CountryCode`). In each case, this relationship applies:

```
{cl.CountryCode} -> {co.Code}
```

`cl.CountryCode` is itself functionally dependent on {`cl.CountryCode`, `cl.Language`} which is a primary key.

If in the join result `co.Code` is `NULL`-complemented, `co.Name` is as well. If `co.Code` is not `NULL`-complemented, then because `co.Code` is a primary key, it determines `co.Name`. Therefore, in all cases:

```
{co.Code} -> {co.Name}
```

Which yields:

```
{cl.CountryCode, cl.Language} -> {cl.*,co.*}
```

As a result, the query is valid.

However, suppose that the tables are swapped, as in this query:

```
SELECT co.Name, cl.Language,
cl.Percentage * co.Population/100.0 AS SpokenBy
FROM country co LEFT JOIN countrylanguage cl
ON cl.CountryCode = co.Code
GROUP BY cl.CountryCode, cl.Language;
```

Now this relationship does *not* apply:

```
{cl.CountryCode, cl.Language} -> {cl.*,co.*}
```

Indeed, all `NULL`-complemented rows made for `cl` is put into a single group (they have both `GROUP BY` columns equal to `NULL`), and inside this group the value of `co.Name` can vary. The query is invalid and MySQL rejects it.

Functional dependence in outer joins is thus linked to whether determinant columns belong to the left or right side of the `LEFT JOIN`. Determination of functional dependence becomes more complex if there are nested outer joins or the join condition does not consist entirely of equality comparisons.

## Functional Dependencies and Views

Suppose that a view on countries produces their code, their name in uppercase, and how many different official languages they have:

```
CREATE VIEW country2 AS
SELECT co.Code, UPPER(co.Name) AS UpperName,
COUNT(cl.Language) AS OfficialLanguages
FROM country AS co JOIN countrylanguage AS cl
ON cl.CountryCode = co.Code
WHERE cl.isOfficial = 'T'
GROUP BY co.Code;
```

This definition is valid because:

```
{co.Code} -> {co.*}
```

In the view result, the first selected column is `co.Code`, which is also the group column and thus determines all other selected expressions:

```
{country2.Code} -> {country2.*}
```

MySQL understands this and uses this information, as described following.

This query displays countries, how many different official languages they have, and how many cities they have, by joining the view with the `city` table:

```
SELECT co2.Code, co2.UpperName, co2.OfficialLanguages,
COUNT(*) AS Cities
FROM country2 AS co2 JOIN city ci
ON ci.CountryCode = co2.Code
GROUP BY co2.Code;
```

This query is valid because, as seen previously:

```
{co2.Code} -> {co2.*}
```

MySQL is able to discover a functional dependency in the result of a view and use that to validate a query which uses the view. The same would be true if `country2` were a derived table (or common table expression), as in:

```
SELECT co2.Code, co2.UpperName, co2.OfficialLanguages,
COUNT(*) AS Cities
FROM
(
 SELECT co.Code, UPPER(co.Name) AS UpperName,
 COUNT(cl.Language) AS OfficialLanguages
 FROM country AS co JOIN countrylanguage AS cl
```

```
 ON cl.CountryCode=co.Code
 WHERE cl.isOfficial='T'
 GROUP BY co.Code
) AS co2
JOIN city ci ON ci.CountryCode = co2.Code
GROUP BY co2.Code;
```

### Combinations of Functional Dependencies

MySQL is able to combine all of the preceding types of functional dependencies (key based, equality based, view based) to validate more complex queries.

# 12.21 Window Functions

MySQL supports window functions that, for each row from a query, perform a calculation using rows related to that row. The following sections discuss how to use window functions, including descriptions of the `OVER` and `WINDOW` clauses. The first section provides descriptions of the nonaggregate window functions. For descriptions of the aggregate window functions, see Section 12.20.1, “Aggregate Function Descriptions”.

For information about optimization and window functions, see Section 8.2.1.21, “Window Function Optimization”.

## 12.21.1 Window Function Descriptions

This section describes nonaggregate window functions that, for each row from a query, perform a calculation using rows related to that row. Most aggregate functions also can be used as window functions; see Section 12.20.1, “Aggregate Function Descriptions”.

For window function usage information and examples, and definitions of terms such as the `OVER` clause, window, partition, frame, and peer, see Section 12.21.2, “Window Function Concepts and Syntax”.

**Table 12.26 Window Functions**

| Name | Description |
|---|---|
| `CUME_DIST()` | Cumulative distribution value |
| `DENSE_RANK()` | Rank of current row within its partition, without gaps |
| `FIRST_VALUE()` | Value of argument from first row of window frame |
| `LAG()` | Value of argument from row lagging current row within partition |
| `LAST_VALUE()` | Value of argument from last row of window frame |
| `LEAD()` | Value of argument from row leading current row within partition |
| `NTH_VALUE()` | Value of argument from N-th row of window frame |
| `NTILE()` | Bucket number of current row within its partition. |
| `PERCENT_RANK()` | Percentage rank value |
| `RANK()` | Rank of current row within its partition, with gaps |
| `ROW_NUMBER()` | Number of current row within its partition |

In the following function descriptions, *`over_clause`* represents the `OVER` clause, described in Section 12.21.2, “Window Function Concepts and Syntax”. Some window functions permit a *`null_treatment`* clause that specifies how to handle `NULL` values when calculating results. This clause is optional. It is part of the SQL standard, but the MySQL implementation permits only `RESPECT NULLS` (which is also the default). This means that `NULL` values are considered when calculating results. `IGNORE NULLS` is parsed, but produces an error.

- `CUME_DIST()` *`over_clause`*

  Returns the cumulative distribution of a value within a group of values; that is, the percentage of partition values less than or equal to the value in the current row. This represents the number of rows preceding or peer with the current row in the window ordering of the window partition divided by the total number of rows in the window partition. Return values range from 0 to 1.

  This function should be used with `ORDER BY` to sort partition rows into the desired order. Without `ORDER BY`, all rows are peers and have value *`N`*/*`N`* = 1, where *`N`* is the partition size.

  *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

  The following query shows, for the set of values in the `val` column, the `CUME_DIST()` value for each row, as well as the percentage rank value returned by the similar `PERCENT_RANK()` function. For reference, the query also displays row numbers using `ROW_NUMBER()`:

```
mysql> SELECT
         val,
         ROW_NUMBER()   OVER w AS 'row_number',
         CUME_DIST()    OVER w AS 'cume_dist',
         PERCENT_RANK() OVER w AS 'percent_rank'
       FROM numbers
       WINDOW w AS (ORDER BY val);
+------+------------+--------------------+--------------+
| val  | row_number | cume_dist          | percent_rank |
+------+------------+--------------------+--------------+
|    1 |          1 | 0.2222222222222222 |            0 |
|    1 |          2 | 0.2222222222222222 |            0 |
|    2 |          3 | 0.3333333333333333 |         0.25 |
|    3 |          4 | 0.6666666666666666 |        0.375 |
|    3 |          5 | 0.6666666666666666 |        0.375 |
|    3 |          6 | 0.6666666666666666 |        0.375 |
|    4 |          7 | 0.8888888888888888 |         0.75 |
|    4 |          8 | 0.8888888888888888 |         0.75 |
|    5 |          9 |                  1 |            1 |
+------+------------+--------------------+--------------+
```

- `DENSE_RANK()` *`over_clause`*

  Returns the rank of the current row within its partition, without gaps. Peers are considered ties and receive the same rank. This function assigns consecutive ranks to peer groups; the result is that groups of size greater than one do not produce noncontiguous rank numbers. For an example, see the `RANK()` function description.

  This function should be used with `ORDER BY` to sort partition rows into the desired order. Without `ORDER BY`, all rows are peers.

  *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

- `FIRST_VALUE(`*`expr`*`)` [*`null_treatment`*] *`over_clause`*

  Returns the value of *`expr`* from the first row of the window frame.

  *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”. *`null_treatment`* is as described in the section introduction.

  The following query demonstrates `FIRST_VALUE()`, `LAST_VALUE()`, and two instances of `NTH_VALUE()`:

```
mysql> SELECT
         time, subject, val,
         FIRST_VALUE(val)  OVER w AS 'first',
         LAST_VALUE(val)   OVER w AS 'last',
         NTH_VALUE(val, 2) OVER w AS 'second',
         NTH_VALUE(val, 4) OVER w AS 'fourth'
       FROM observations
```

```
       WINDOW w AS (PARTITION BY subject ORDER BY time
                    ROWS UNBOUNDED PRECEDING);
+----------+---------+------+-------+------+--------+--------+
| time     | subject | val  | first | last | second | fourth |
+----------+---------+------+-------+------+--------+--------+
| 07:00:00 | st113   |   10 |    10 |   10 |   NULL |   NULL |
| 07:15:00 | st113   |    9 |    10 |    9 |      9 |   NULL |
| 07:30:00 | st113   |   25 |    10 |   25 |      9 |   NULL |
| 07:45:00 | st113   |   20 |    10 |   20 |      9 |     20 |
| 07:00:00 | xh458   |    0 |     0 |    0 |   NULL |   NULL |
| 07:15:00 | xh458   |   10 |     0 |   10 |     10 |   NULL |
| 07:30:00 | xh458   |    5 |     0 |    5 |     10 |   NULL |
| 07:45:00 | xh458   |   30 |     0 |   30 |     10 |     30 |
| 08:00:00 | xh458   |   25 |     0 |   25 |     10 |     30 |
+----------+---------+------+-------+------+--------+--------+
```

Each function uses the rows in the current frame, which, per the window definition shown, extends from the first partition row to the current row. For the `NTH_VALUE()` calls, the current frame does not always include the requested row; in such cases, the return value is `NULL`.

- `LAG(`*`expr [, N[, default]]`*`)` [*`null_treatment`*] *`over_clause`*

  Returns the value of *`expr`* from the row that lags (precedes) the current row by *`N`* rows within its partition. If there is no such row, the return value is *`default`*. For example, if *`N`* is 3, the return value is *`default`* for the first three rows. If *`N`* or *`default`* are missing, the defaults are 1 and `NULL`, respectively.

  *`N`* must be a literal nonnegative integer. If *`N`* is 0, *`expr`* is evaluated for the current row.

  Beginning with MySQL 8.0.22, *`N`* cannot be `NULL`. In addition, it must now be an integer in the range `1` to $2^{63}$, inclusive, in any of the following forms:

  - an unsigned integer constant literal
  - a positional parameter marker (`?`)
  - a user-defined variable
  - a local variable in a stored routine

  *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”. *`null_treatment`* is as described in the section introduction.

  `LAG()` (and the similar `LEAD()` function) are often used to compute differences between rows. The following query shows a set of time-ordered observations and, for each one, the `LAG()` and `LEAD()` values from the adjoining rows, as well as the differences between the current and adjoining rows:

```
mysql> SELECT
         t, val,
         LAG(val)        OVER w AS 'lag',
         LEAD(val)       OVER w AS 'lead',
         val - LAG(val)  OVER w AS 'lag diff',
         val - LEAD(val) OVER w AS 'lead diff'
       FROM series
       WINDOW w AS (ORDER BY t);
+----------+------+------+------+----------+-----------+
| t        | val  | lag  | lead | lag diff | lead diff |
+----------+------+------+------+----------+-----------+
| 12:00:00 |  100 | NULL |  125 |     NULL |       -25 |
| 13:00:00 |  125 |  100 |  132 |       25 |        -7 |
| 14:00:00 |  132 |  125 |  145 |        7 |       -13 |
| 15:00:00 |  145 |  132 |  140 |       13 |         5 |
| 16:00:00 |  140 |  145 |  150 |       -5 |       -10 |
| 17:00:00 |  150 |  140 |  200 |       10 |       -50 |
| 18:00:00 |  200 |  150 | NULL |       50 |      NULL |
```

```
+----------+------+------+------+----------+-----------+
```

In the example, the `LAG()` and `LEAD()` calls use the default *`N`* and *`default`* values of 1 and `NULL`, respectively.

The first row shows what happens when there is no previous row for `LAG()`: The function returns the *`default`* value (in this case, `NULL`). The last row shows the same thing when there is no next row for `LEAD()`.

`LAG()` and `LEAD()` also serve to compute sums rather than differences. Consider this data set, which contains the first few numbers of the Fibonacci series:

```
mysql> SELECT n FROM fib ORDER BY n;
+------+
| n    |
+------+
|    1 |
|    1 |
|    2 |
|    3 |
|    5 |
|    8 |
+------+
```

The following query shows the `LAG()` and `LEAD()` values for the rows adjacent to the current row. It also uses those functions to add to the current row value the values from the preceding and following rows. The effect is to generate the next number in the Fibonacci series, and the next number after that:

```
mysql> SELECT
         n,
         LAG(n, 1, 0)      OVER w AS 'lag',
         LEAD(n, 1, 0)     OVER w AS 'lead',
         n + LAG(n, 1, 0)  OVER w AS 'next_n',
         n + LEAD(n, 1, 0) OVER w AS 'next_next_n'
       FROM fib
       WINDOW w AS (ORDER BY n);
+------+------+------+--------+-------------+
| n    | lag  | lead | next_n | next_next_n |
+------+------+------+--------+-------------+
|    1 |    0 |    1 |      1 |           2 |
|    1 |    1 |    2 |      2 |           3 |
|    2 |    1 |    3 |      3 |           5 |
|    3 |    2 |    5 |      5 |           8 |
|    5 |    3 |    8 |      8 |          13 |
|    8 |    5 |    0 |     13 |           8 |
+------+------+------+--------+-------------+
```

One way to generate the initial set of Fibonacci numbers is to use a recursive common table expression. For an example, see Fibonacci Series Generation.

Beginning with MySQL 8.0.22, you cannot use a negative value for the rows argument of this function.

- `LAST_VALUE(`*`expr`*`)` [*`null_treatment`*] *`over_clause`*

  Returns the value of *`expr`* from the last row of the window frame.

  *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”. *`null_treatment`* is as described in the section introduction.

  For an example, see the `FIRST_VALUE()` function description.

- `LEAD(expr [, N[, default]])` [`null_treatment`] `over_clause`

  Returns the value of `expr` from the row that leads (follows) the current row by `N` rows within its partition. If there is no such row, the return value is `default`. For example, if `N` is 3, the return value is `default` for the last three rows. If `N` or `default` are missing, the defaults are 1 and `NULL`, respectively.

  `N` must be a literal nonnegative integer. If `N` is 0, `expr` is evaluated for the current row.

  Beginning with MySQL 8.0.22, `N` cannot be `NULL`. In addition, it must now be an integer in the range `1` to $2^{63}$, inclusive, in any of the following forms:

  - an unsigned integer constant literal
  - a positional parameter marker (`?`)
  - a user-defined variable
  - a local variable in a stored routine

  `over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”. `null_treatment` is as described in the section introduction.

  For an example, see the `LAG()` function description.

  In MySQL 8.0.22 and later, use of a negative value for the rows argument of this function is not permitted.

- `NTH_VALUE(expr, N)` [`from_first_last`] [`null_treatment`] `over_clause`

  Returns the value of `expr` from the `N`-th row of the window frame. If there is no such row, the return value is `NULL`.

  `N` must be a literal positive integer.

  `from_first_last` is part of the SQL standard, but the MySQL implementation permits only `FROM FIRST` (which is also the default). This means that calculations begin at the first row of the window. `FROM LAST` is parsed, but produces an error. To obtain the same effect as `FROM LAST` (begin calculations at the last row of the window), use `ORDER BY` to sort in reverse order.

  `over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”. `null_treatment` is as described in the section introduction.

  For an example, see the `FIRST_VALUE()` function description.

  In MySQL 8.0.22 and later, you cannot use `NULL` for the row argument of this function.

- `NTILE(N) over_clause`

  Divides a partition into *`N`* groups (buckets), assigns each row in the partition its bucket number, and returns the bucket number of the current row within its partition. For example, if *`N`* is 4, `NTILE()` divides rows into four buckets. If *`N`* is 100, `NTILE()` divides rows into 100 buckets.

  *`N`* must be a literal positive integer. Bucket number return values range from 1 to *`N`*.

  Beginning with MySQL 8.0.22, *`N`* cannot be `NULL`. In addition, it must be an integer in the range `1` to $2^{63}$, inclusive, in any of the following forms:

  - an unsigned integer constant literal
  - a positional parameter marker (`?`)
  - a user-defined variable
  - a local variable in a stored routine

  This function should be used with `ORDER BY` to sort partition rows into the desired order.

  *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

  The following query shows, for the set of values in the `val` column, the percentile values resulting from dividing the rows into two or four groups. For reference, the query also displays row numbers using `ROW_NUMBER()`:

```
mysql> SELECT
         val,
         ROW_NUMBER() OVER w AS 'row_number',
         NTILE(2)     OVER w AS 'ntile2',
         NTILE(4)     OVER w AS 'ntile4'
       FROM numbers
       WINDOW w AS (ORDER BY val);
+------+------------+--------+--------+
| val  | row_number | ntile2 | ntile4 |
+------+------------+--------+--------+
|    1 |          1 |      1 |      1 |
|    1 |          2 |      1 |      1 |
|    2 |          3 |      1 |      1 |
|    3 |          4 |      1 |      2 |
|    3 |          5 |      1 |      2 |
|    3 |          6 |      2 |      3 |
|    4 |          7 |      2 |      3 |
|    4 |          8 |      2 |      4 |
|    5 |          9 |      2 |      4 |
+------+------------+--------+--------+
```

  Beginning with MySQL 8.0.22, the construct `NTILE(NULL)` is no longer permitted.

- `PERCENT_RANK() over_clause`

  Returns the percentage of partition values less than the value in the current row, excluding the highest value. Return values range from 0 to 1 and represent the row relative rank, calculated as the result of this formula, where *`rank`* is the row rank and *`rows`* is the number of partition rows:

```
(rank - 1) / (rows - 1)
```

  This function should be used with `ORDER BY` to sort partition rows into the desired order. Without `ORDER BY`, all rows are peers.

  *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

  For an example, see the `CUME_DIST()` function description.

- `RANK() over_clause`

  Returns the rank of the current row within its partition, with gaps. Peers are considered ties and receive the same rank. This function does not assign consecutive ranks to peer groups if groups of size greater than one exist; the result is noncontiguous rank numbers.

  This function should be used with `ORDER BY` to sort partition rows into the desired order. Without `ORDER BY`, all rows are peers.

  *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

  The following query shows the difference between `RANK()`, which produces ranks with gaps, and `DENSE_RANK()`, which produces ranks without gaps. The query shows rank values for each member of a set of values in the `val` column, which contains some duplicates. `RANK()` assigns peers (the duplicates) the same rank value, and the next greater value has a rank higher by the number of peers minus one. `DENSE_RANK()` also assigns peers the same rank value, but the next higher value has a rank one greater. For reference, the query also displays row numbers using `ROW_NUMBER()`:

```
mysql> SELECT
         val,
         ROW_NUMBER() OVER w AS 'row_number',
         RANK()       OVER w AS 'rank',
         DENSE_RANK() OVER w AS 'dense_rank'
       FROM numbers
       WINDOW w AS (ORDER BY val);
+------+------------+------+------------+
| val  | row_number | rank | dense_rank |
+------+------------+------+------------+
|    1 |          1 |    1 |          1 |
|    1 |          2 |    1 |          1 |
|    2 |          3 |    3 |          2 |
|    3 |          4 |    4 |          3 |
|    3 |          5 |    4 |          3 |
|    3 |          6 |    4 |          3 |
|    4 |          7 |    7 |          4 |
|    4 |          8 |    7 |          4 |
|    5 |          9 |    9 |          5 |
+------+------------+------+------------+
```

- `ROW_NUMBER() over_clause`

  Returns the number of the current row within its partition. Rows numbers range from 1 to the number of partition rows.

  `ORDER BY` affects the order in which rows are numbered. Without `ORDER BY`, row numbering is nondeterministic.

  `ROW_NUMBER()` assigns peers different row numbers. To assign peers the same value, use `RANK()` or `DENSE_RANK()`. For an example, see the `RANK()` function description.

  *`over_clause`* is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

## 12.21.2 Window Function Concepts and Syntax

This section describes how to use window functions. Examples use the same sales information data set as found in the discussion of the `GROUPING()` function in Section 12.20.2, “GROUP BY Modifiers”:

```
mysql> SELECT * FROM sales ORDER BY country, year, product;
+------+---------+------------+--------+
| year | country | product    | profit |
+------+---------+------------+--------+
| 2000 | Finland | Computer   |   1500 |
| 2000 | Finland | Phone      |    100 |
| 2001 | Finland | Phone      |     10 |
| 2000 | India   | Calculator |     75 |
| 2000 | India   | Calculator |     75 |
```

```
| 2000 | India   | Computer   |   1200 |
| 2000 | USA     | Calculator |     75 |
| 2000 | USA     | Computer   |   1500 |
| 2001 | USA     | Calculator |     50 |
| 2001 | USA     | Computer   |   1500 |
| 2001 | USA     | Computer   |   1200 |
| 2001 | USA     | TV         |    150 |
| 2001 | USA     | TV         |    100 |
+------+---------+------------+--------+
```

A window function performs an aggregate-like operation on a set of query rows. However, whereas an aggregate operation groups query rows into a single result row, a window function produces a result for each query row:

- The row for which function evaluation occurs is called the current row.
- The query rows related to the current row over which function evaluation occurs comprise the window for the current row.

For example, using the sales information table, these two queries perform aggregate operations that produce a single global sum for all rows taken as a group, and sums grouped per country:

```
mysql> SELECT SUM(profit) AS total_profit
       FROM sales;
+--------------+
| total_profit |
+--------------+
|         7535 |
+--------------+
mysql> SELECT country, SUM(profit) AS country_profit
       FROM sales
       GROUP BY country
       ORDER BY country;
+---------+----------------+
| country | country_profit |
+---------+----------------+
| Finland |           1610 |
| India   |           1350 |
| USA     |           4575 |
+---------+----------------+
```

By contrast, window operations do not collapse groups of query rows to a single output row. Instead, they produce a result for each row. Like the preceding queries, the following query uses `SUM()`, but this time as a window function:

```
mysql> SELECT
         year, country, product, profit,
         SUM(profit) OVER() AS total_profit,
         SUM(profit) OVER(PARTITION BY country) AS country_profit
       FROM sales
       ORDER BY country, year, product, profit;
+------+---------+------------+--------+--------------+----------------+
| year | country | product    | profit | total_profit | country_profit |
+------+---------+------------+--------+--------------+----------------+
| 2000 | Finland | Computer   |   1500 |         7535 |           1610 |
| 2000 | Finland | Phone      |    100 |         7535 |           1610 |
| 2001 | Finland | Phone      |     10 |         7535 |           1610 |
| 2000 | India   | Calculator |     75 |         7535 |           1350 |
| 2000 | India   | Calculator |     75 |         7535 |           1350 |
| 2000 | India   | Computer   |   1200 |         7535 |           1350 |
| 2000 | USA     | Calculator |     75 |         7535 |           4575 |
| 2000 | USA     | Computer   |   1500 |         7535 |           4575 |
| 2001 | USA     | Calculator |     50 |         7535 |           4575 |
| 2001 | USA     | Computer   |   1200 |         7535 |           4575 |
| 2001 | USA     | Computer   |   1500 |         7535 |           4575 |
| 2001 | USA     | TV         |    100 |         7535 |           4575 |
| 2001 | USA     | TV         |    150 |         7535 |           4575 |
+------+---------+------------+--------+--------------+----------------+
```

Each window operation in the query is signified by inclusion of an `OVER` clause that specifies how to partition query rows into groups for processing by the window function:

- The first `OVER` clause is empty, which treats the entire set of query rows as a single partition. The window function thus produces a global sum, but does so for each row.
- The second `OVER` clause partitions rows by country, producing a sum per partition (per country). The function produces this sum for each partition row.

Window functions are permitted only in the select list and `ORDER BY` clause. Query result rows are determined from the `FROM` clause, after `WHERE`, `GROUP BY`, and `HAVING` processing, and windowing execution occurs before `ORDER BY`, `LIMIT`, and `SELECT DISTINCT`.

The `OVER` clause is permitted for many aggregate functions, which therefore can be used as window or nonwindow functions, depending on whether the `OVER` clause is present or absent:

```
AVG()
BIT_AND()
BIT_OR()
BIT_XOR()
COUNT()
JSON_ARRAYAGG()
JSON_OBJECTAGG()
MAX()
MIN()
STDDEV_POP(), STDDEV(), STD()
STDDEV_SAMP()
SUM()
VAR_POP(), VARIANCE()
VAR_SAMP()
```

For details about each aggregate function, see Section 12.20.1, “Aggregate Function Descriptions”.

MySQL also supports nonaggregate functions that are used only as window functions. For these, the `OVER` clause is mandatory:

```
CUME_DIST()
DENSE_RANK()
FIRST_VALUE()
LAG()
LAST_VALUE()
LEAD()
NTH_VALUE()
NTILE()
PERCENT_RANK()
RANK()
ROW_NUMBER()
```

For details about each nonaggregate function, see Section 12.21.1, “Window Function Descriptions”.

As an example of one of those nonaggregate window functions, this query uses `ROW_NUMBER()`, which produces the row number of each row within its partition. In this case, rows are numbered per country. By default, partition rows are unordered and row numbering is nondeterministic. To sort partition rows, include an `ORDER BY` clause within the window definition. The query uses unordered and ordered partitions (the `row_num1` and `row_num2` columns) to illustrate the difference between omitting and including `ORDER BY`:

```
mysql> SELECT
         year, country, product, profit,
         ROW_NUMBER() OVER(PARTITION BY country) AS row_num1,
         ROW_NUMBER() OVER(PARTITION BY country ORDER BY year, product) AS row_num2
       FROM sales;
+------+---------+------------+--------+----------+----------+
| year | country | product    | profit | row_num1 | row_num2 |
+------+---------+------------+--------+----------+----------+
| 2000 | Finland | Computer   |   1500 |        2 |        1 |
| 2000 | Finland | Phone      |    100 |        1 |        2 |
```

```
| 2001 | Finland | Phone      |     10 |        3 |        3 |
| 2000 | India   | Calculator |     75 |        2 |        1 |
| 2000 | India   | Calculator |     75 |        3 |        2 |
| 2000 | India   | Computer   |   1200 |        1 |        3 |
| 2000 | USA     | Calculator |     75 |        5 |        1 |
| 2000 | USA     | Computer   |   1500 |        4 |        2 |
| 2001 | USA     | Calculator |     50 |        2 |        3 |
| 2001 | USA     | Computer   |   1500 |        3 |        4 |
| 2001 | USA     | Computer   |   1200 |        7 |        5 |
| 2001 | USA     | TV         |    150 |        1 |        6 |
| 2001 | USA     | TV         |    100 |        6 |        7 |
+------+---------+------------+--------+----------+----------+
```

As mentioned previously, to use a window function (or treat an aggregate function as a window function), include an `OVER` clause following the function call. The `OVER` clause has two forms:

```
over_clause:
    {OVER (window_spec) | OVER window_name}
```

Both forms define how the window function should process query rows. They differ in whether the window is defined directly in the `OVER` clause, or supplied by a reference to a named window defined elsewhere in the query:

- In the first case, the window specification appears directly in the `OVER` clause, between the parentheses.
- In the second case, *`window_name`* is the name for a window specification defined by a `WINDOW` clause elsewhere in the query. For details, see Section 12.21.4, “Named Windows”.

For `OVER (window_spec)` syntax, the window specification has several parts, all optional:

```
window_spec:
    [window_name] [partition_clause] [order_clause] [frame_clause]
```

If `OVER()` is empty, the window consists of all query rows and the window function computes a result using all rows. Otherwise, the clauses present within the parentheses determine which query rows are used to compute the function result and how they are partitioned and ordered:

- *`window_name`*: The name of a window defined by a `WINDOW` clause elsewhere in the query. If *`window_name`* appears by itself within the `OVER` clause, it completely defines the window. If partitioning, ordering, or framing clauses are also given, they modify interpretation of the named window. For details, see Section 12.21.4, “Named Windows”.
- *`partition_clause`*: A `PARTITION BY` clause indicates how to divide the query rows into groups. The window function result for a given row is based on the rows of the partition that contains the row. If `PARTITION BY` is omitted, there is a single partition consisting of all query rows.

  

  **Note**

  Partitioning for window functions differs from table partitioning. For information about table partitioning, see Chapter 24, *Partitioning*.

  *`partition_clause`* has this syntax:

  ```
  partition_clause:
      PARTITION BY expr [, expr] ...
  ```

  Standard SQL requires `PARTITION BY` to be followed by column names only. A MySQL extension is to permit expressions, not just column names. For example, if a table contains a `TIMESTAMP` column named `ts`, standard SQL permits `PARTITION BY ts` but not `PARTITION BY HOUR(ts)`, whereas MySQL permits both.

- *`order_clause`*: An `ORDER BY` clause indicates how to sort rows in each partition. Partition rows that are equal according to the `ORDER BY` clause are considered peers. If `ORDER BY` is omitted, partition rows are unordered, with no processing order implied, and all partition rows are peers.

*order_clause* has this syntax:

```
order_clause:
    ORDER BY expr [ASC|DESC] [, expr [ASC|DESC]] ...
```

Each ORDER BY expression optionally can be followed by ASC or DESC to indicate sort direction. The default is ASC if no direction is specified. NULL values sort first for ascending sorts, last for descending sorts.

An ORDER BY in a window definition applies within individual partitions. To sort the result set as a whole, include an ORDER BY at the query top level.

- *frame_clause*: A frame is a subset of the current partition and the frame clause specifies how to define the subset. The frame clause has many subclauses of its own. For details, see Section 12.21.3, “Window Function Frame Specification”.

## 12.21.3 Window Function Frame Specification

The definition of a window used with a window function can include a frame clause. A frame is a subset of the current partition and the frame clause specifies how to define the subset.

Frames are determined with respect to the current row, which enables a frame to move within a partition depending on the location of the current row within its partition. Examples:

- By defining a frame to be all rows from the partition start to the current row, you can compute running totals for each row.
- By defining a frame as extending *N* rows on either side of the current row, you can compute rolling averages.

The following query demonstrates the use of moving frames to compute running totals within each group of time-ordered level values, as well as rolling averages computed from the current row and the rows that immediately precede and follow it:

```
mysql> SELECT
         time, subject, val,
         SUM(val) OVER (PARTITION BY subject ORDER BY time
                        ROWS UNBOUNDED PRECEDING)
           AS running_total,
         AVG(val) OVER (PARTITION BY subject ORDER BY time
                        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
           AS running_average
       FROM observations;
+----------+---------+------+---------------+-----------------+
| time     | subject | val  | running_total | running_average |
+----------+---------+------+---------------+-----------------+
| 07:00:00 | st113   |   10 |            10 |          9.5000 |
| 07:15:00 | st113   |    9 |            19 |         14.6667 |
| 07:30:00 | st113   |   25 |            44 |         18.0000 |
| 07:45:00 | st113   |   20 |            64 |         22.5000 |
| 07:00:00 | xh458   |    0 |             0 |          5.0000 |
| 07:15:00 | xh458   |   10 |            10 |          5.0000 |
| 07:30:00 | xh458   |    5 |            15 |         15.0000 |
| 07:45:00 | xh458   |   30 |            45 |         20.0000 |
| 08:00:00 | xh458   |   25 |            70 |         27.5000 |
+----------+---------+------+---------------+-----------------+
```

For the running_average column, there is no frame row preceding the first one or following the last. In these cases, AVG() computes the average of the rows that are available.

Aggregate functions used as window functions operate on rows in the current row frame, as do these nonaggregate window functions:

```
FIRST_VALUE()
LAST_VALUE()
NTH_VALUE()
```

Standard SQL specifies that window functions that operate on the entire partition should have no frame clause. MySQL permits a frame clause for such functions but ignores it. These functions use the entire partition even if a frame is specified:

```
CUME_DIST()
DENSE_RANK()
LAG()
LEAD()
NTILE()
PERCENT_RANK()
RANK()
ROW_NUMBER()
```

The frame clause, if given, has this syntax:

```
frame_clause:
    frame_units frame_extent

frame_units:
    {ROWS | RANGE}
```

In the absence of a frame clause, the default frame depends on whether an `ORDER BY` clause is present, as described later in this section.

The *`frame_units`* value indicates the type of relationship between the current row and frame rows:

- `ROWS`: The frame is defined by beginning and ending row positions. Offsets are differences in row numbers from the current row number.
- `RANGE`: The frame is defined by rows within a value range. Offsets are differences in row values from the current row value.

The *`frame_extent`* value indicates the start and end points of the frame. You can specify just the start of the frame (in which case the current row is implicitly the end) or use `BETWEEN` to specify both frame endpoints:

```
frame_extent:
    {frame_start | frame_between}

frame_between:
    BETWEEN frame_start AND frame_end

frame_start, frame_end: {
    CURRENT ROW
  | UNBOUNDED PRECEDING
  | UNBOUNDED FOLLOWING
  | expr PRECEDING
  | expr FOLLOWING
}
```

With `BETWEEN` syntax, *`frame_start`* must not occur later than *`frame_end`*.

The permitted *`frame_start`* and *`frame_end`* values have these meanings:

- `CURRENT ROW`: For `ROWS`, the bound is the current row. For `RANGE`, the bound is the peers of the current row.
- `UNBOUNDED PRECEDING`: The bound is the first partition row.
- `UNBOUNDED FOLLOWING`: The bound is the last partition row.
- *`expr`* `PRECEDING`: For `ROWS`, the bound is *`expr`* rows before the current row. For `RANGE`, the bound is the rows with values equal to the current row value minus *`expr`*; if the current row value is `NULL`, the bound is the peers of the row.

  For *`expr`* `PRECEDING` (and *`expr`* `FOLLOWING`), *`expr`* can be a `?` parameter marker (for use in a prepared statement), a nonnegative numeric literal, or a temporal interval of the form `INTERVAL`

*val unit*. For INTERVAL expressions, *val* specifies nonnegative interval value, and *unit* is a keyword indicating the units in which the value should be interpreted. (For details about the permitted *units* specifiers, see the description of the DATE_ADD() function in Section 12.7, “Date and Time Functions”.)

RANGE on a numeric or temporal *expr* requires ORDER BY on a numeric or temporal expression, respectively.

Examples of valid *expr* PRECEDING and *expr* FOLLOWING indicators:

```
10 PRECEDING
INTERVAL 5 DAY PRECEDING
5 FOLLOWING
INTERVAL '2:30' MINUTE_SECOND FOLLOWING
```

- *expr* FOLLOWING: For ROWS, the bound is *expr* rows after the current row. For RANGE, the bound is the rows with values equal to the current row value plus *expr*; if the current row value is NULL, the bound is the peers of the row.

  For permitted values of *expr*, see the description of *expr* PRECEDING.

The following query demonstrates FIRST_VALUE(), LAST_VALUE(), and two instances of NTH_VALUE():

```
mysql> SELECT
         time, subject, val,
         FIRST_VALUE(val)  OVER w AS 'first',
         LAST_VALUE(val)   OVER w AS 'last',
         NTH_VALUE(val, 2) OVER w AS 'second',
         NTH_VALUE(val, 4) OVER w AS 'fourth'
       FROM observations
       WINDOW w AS (PARTITION BY subject ORDER BY time
                    ROWS UNBOUNDED PRECEDING);
+----------+---------+------+-------+------+--------+--------+
| time     | subject | val  | first | last | second | fourth |
+----------+---------+------+-------+------+--------+--------+
| 07:00:00 | st113   |   10 |    10 |   10 |   NULL |   NULL |
| 07:15:00 | st113   |    9 |    10 |    9 |      9 |   NULL |
| 07:30:00 | st113   |   25 |    10 |   25 |      9 |   NULL |
| 07:45:00 | st113   |   20 |    10 |   20 |      9 |     20 |
| 07:00:00 | xh458   |    0 |     0 |    0 |   NULL |   NULL |
| 07:15:00 | xh458   |   10 |     0 |   10 |     10 |   NULL |
| 07:30:00 | xh458   |    5 |     0 |    5 |     10 |   NULL |
| 07:45:00 | xh458   |   30 |     0 |   30 |     10 |     30 |
| 08:00:00 | xh458   |   25 |     0 |   25 |     10 |     30 |
+----------+---------+------+-------+------+--------+--------+
```

Each function uses the rows in the current frame, which, per the window definition shown, extends from the first partition row to the current row. For the NTH_VALUE() calls, the current frame does not always include the requested row; in such cases, the return value is NULL.

In the absence of a frame clause, the default frame depends on whether an ORDER BY clause is present:

- With ORDER BY: The default frame includes rows from the partition start through the current row, including all peers of the current row (rows equal to the current row according to the ORDER BY clause). The default is equivalent to this frame specification:

  ```
  RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  ```

- Without ORDER BY: The default frame includes all partition rows (because, without ORDER BY, all partition rows are peers). The default is equivalent to this frame specification:

  ```
  RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
  ```

Because the default frame differs depending on presence or absence of ORDER BY, adding ORDER BY to a query to get deterministic results may change the results. (For example, the values produced by

SUM() might change.) To obtain the same results but ordered per ORDER BY, provide an explicit frame specification to be used regardless of whether ORDER BY is present.

The meaning of a frame specification can be nonobvious when the current row value is NULL. Assuming that to be the case, these examples illustrate how various frame specifications apply:

- ORDER BY X ASC RANGE BETWEEN 10 FOLLOWING AND 15 FOLLOWING

  The frame starts at NULL and stops at NULL, thus includes only rows with value NULL.

- ORDER BY X ASC RANGE BETWEEN 10 FOLLOWING AND UNBOUNDED FOLLOWING

  The frame starts at NULL and stops at the end of the partition. Because an ASC sort puts NULL values first, the frame is the entire partition.

- ORDER BY X DESC RANGE BETWEEN 10 FOLLOWING AND UNBOUNDED FOLLOWING

  The frame starts at NULL and stops at the end of the partition. Because a DESC sort puts NULL values last, the frame is only the NULL values.

- ORDER BY X ASC RANGE BETWEEN 10 PRECEDING AND UNBOUNDED FOLLOWING

  The frame starts at NULL and stops at the end of the partition. Because an ASC sort puts NULL values first, the frame is the entire partition.

- ORDER BY X ASC RANGE BETWEEN 10 PRECEDING AND 10 FOLLOWING

  The frame starts at NULL and stops at NULL, thus includes only rows with value NULL.

- ORDER BY X ASC RANGE BETWEEN 10 PRECEDING AND 1 PRECEDING

  The frame starts at NULL and stops at NULL, thus includes only rows with value NULL.

- ORDER BY X ASC RANGE BETWEEN UNBOUNDED PRECEDING AND 10 FOLLOWING

  The frame starts at the beginning of the partition and stops at rows with value NULL. Because an ASC sort puts NULL values first, the frame is only the NULL values.

## 12.21.4 Named Windows

Windows can be defined and given names by which to refer to them in OVER clauses. To do this, use a WINDOW clause. If present in a query, the WINDOW clause falls between the positions of the HAVING and ORDER BY clauses, and has this syntax:

```
WINDOW window_name AS (window_spec)
    [, window_name AS (window_spec)] ...
```

For each window definition, *window_name* is the window name, and *window_spec* is the same type of window specification as given between the parentheses of an OVER clause, as described in Section 12.21.2, “Window Function Concepts and Syntax”:

```
window_spec:
    [window_name] [partition_clause] [order_clause] [frame_clause]
```

A WINDOW clause is useful for queries in which multiple OVER clauses would otherwise define the same window. Instead, you can define the window once, give it a name, and refer to the name in the OVER clauses. Consider this query, which defines the same window multiple times:

```
SELECT
  val,
  ROW_NUMBER() OVER (ORDER BY val) AS 'row_number',
  RANK()       OVER (ORDER BY val) AS 'rank',
  DENSE_RANK() OVER (ORDER BY val) AS 'dense_rank'
FROM numbers;
```

The query can be written more simply by using `WINDOW` to define the window once and referring to the window by name in the `OVER` clauses:

```
SELECT
  val,
  ROW_NUMBER() OVER w AS 'row_number',
  RANK()       OVER w AS 'rank',
  DENSE_RANK() OVER w AS 'dense_rank'
FROM numbers
WINDOW w AS (ORDER BY val);
```

A named window also makes it easier to experiment with the window definition to see the effect on query results. You need only modify the window definition in the `WINDOW` clause, rather than multiple `OVER` clause definitions.

If an `OVER` clause uses `OVER (window_name ...)` rather than `OVER window_name`, the named window can be modified by the addition of other clauses. For example, this query defines a window that includes partitioning, and uses `ORDER BY` in the `OVER` clauses to modify the window in different ways:

```
SELECT
  DISTINCT year, country,
  FIRST_VALUE(year) OVER (w ORDER BY year ASC) AS first,
  FIRST_VALUE(year) OVER (w ORDER BY year DESC) AS last
FROM sales
WINDOW w AS (PARTITION BY country);
```

An `OVER` clause can only add properties to a named window, not modify them. If the named window definition includes a partitioning, ordering, or framing property, the `OVER` clause that refers to the window name cannot also include the same kind of property or an error occurs:

- This construct is permitted because the window definition and the referring `OVER` clause do not contain the same kind of properties:

  ```
  OVER (w ORDER BY country)
  ... WINDOW w AS (PARTITION BY country)
  ```

- This construct is not permitted because the `OVER` clause specifies `PARTITION BY` for a named window that already has `PARTITION BY`:

  ```
  OVER (w PARTITION BY year)
  ... WINDOW w AS (PARTITION BY country)
  ```

The definition of a named window can itself begin with a *`window_name`*. In such cases, forward and backward references are permitted, but not cycles:

- This is permitted; it contains forward and backward references but no cycles:

  ```
  WINDOW w1 AS (w2), w2 AS (), w3 AS (w1)
  ```

- This is not permitted because it contains a cycle:

  ```
  WINDOW w1 AS (w2), w2 AS (w3), w3 AS (w1)
  ```

## 12.21.5 Window Function Restrictions

The SQL standard imposes a constraint on window functions that they cannot be used in `UPDATE` or `DELETE` statements to update rows. Using such functions in a subquery of these statements (to select rows) is permitted.

MySQL does not support these window function features:

- `DISTINCT` syntax for aggregate window functions.
- Nested window functions.
- Dynamic frame endpoints that depend on the value of the current row.

The parser recognizes these window constructs which nevertheless are not supported:

- The `GROUPS` frame units specifier is parsed, but produces an error. Only `ROWS` and `RANGE` are supported.
- The `EXCLUDE` clause for frame specification is parsed, but produces an error.
- `IGNORE NULLS` is parsed, but produces an error. Only `RESPECT NULLS` is supported.
- `FROM LAST` is parsed, but produces an error. Only `FROM FIRST` is supported.

As of MySQL 8.0.28, a maximum of 127 windows is supported for a given `SELECT`. Note that a single query may use multiple `SELECT` clauses, and each of these clauses supports up to 127 windows. The number of distinct windows is defined as the sum of the named windows and any implicit windows specified as part of any window function's `OVER` clause. You should also be aware that queries using very large numbers of windows may require increasing the default thread stack size (`thread_stack` system variable).

# 12.22 Performance Schema Functions

As of MySQL 8.0.16, MySQL includes built-in SQL functions that format or retrieve Performance Schema data, and that may be used as equivalents for the corresponding `sys` schema stored functions. The built-in functions can be invoked in any schema and require no qualifier, unlike the `sys` functions, which require either a `sys.` schema qualifier or that `sys` be the current schema.

**Table 12.27 Performance Schema Functions**

| Name | Description | Introduced |
|---|---|---|
| `FORMAT_BYTES()` | Convert byte count to value with units | 8.0.16 |
| `FORMAT_PICO_TIME()` | Convert time in picoseconds to value with units | 8.0.16 |
| `PS_CURRENT_THREAD_ID()` | Performance Schema thread ID for current thread | 8.0.16 |
| `PS_THREAD_ID()` | Performance Schema thread ID for given thread | 8.0.16 |

The built-in functions supersede the corresponding `sys` functions, which are deprecated; expect them to be removed in a future version of MySQL. Applications that use the `sys` functions should be adjusted to use the built-in functions instead, keeping in mind some minor differences between the `sys` functions and the built-in functions. For details about these differences, see the function descriptions in this section.

- `FORMAT_BYTES(`*`count`*`)`

  Given a numeric byte count, converts it to human-readable format and returns a string consisting of a value and a units indicator. The string contains the number of bytes rounded to 2 decimal places and a minimum of 3 significant digits. Numbers less than 1024 bytes are represented as whole numbers and are not rounded. Returns `NULL` if *`count`* is `NULL`.

  The units indicator depends on the size of the byte-count argument as shown in the following table.

| Argument Value | Result Units | Result Units Indicator |
|---|---|---|
| Up to 1023 | bytes | bytes |
| Up to $1024^2 - 1$ | kibibytes | KiB |
| Up to $1024^3 - 1$ | mebibytes | MiB |
| Up to $1024^4 - 1$ | gibibytes | GiB |
| Up to $1024^5 - 1$ | tebibytes | TiB |

| Argument Value | Result Units | Result Units Indicator |
|---|---|---|
| Up to $1024^6 - 1$ | pebibytes | PiB |
| $1024^6$ and up | exbibytes | EiB |

```
mysql> SELECT FORMAT_BYTES(512), FORMAT_BYTES(18446644073709551615);
+------------------+-----------------------------------+
| FORMAT_BYTES(512) | FORMAT_BYTES(18446644073709551615) |
+------------------+-----------------------------------+
|  512 bytes        | 16.00 EiB                         |
+------------------+-----------------------------------+
```

`FORMAT_BYTES()` was added in MySQL 8.0.16. It may be used instead of the `sys` schema `format_bytes()` function, keeping in mind this difference:

- `FORMAT_BYTES()` uses the `EiB` units indicator. `sys.format_bytes()` does not.

- `FORMAT_PICO_TIME(time_val)`

  Given a numeric Performance Schema latency or wait time in picoseconds, converts it to human-readable format and returns a string consisting of a value and a units indicator. The string contains the decimal time rounded to 2 decimal places and a minimum of 3 significant digits. Times under 1 nanosecond are represented as whole numbers and are not rounded.

  If *`time_val`* is `NULL`, this function returns `NULL`.

  The units indicator depends on the size of the time-value argument as shown in the following table.

| Argument Value | Result Units | Result Units Indicator |
|---|---|---|
| Up to $10^3 - 1$ | picoseconds | ps |
| Up to $10^6 - 1$ | nanoseconds | ns |
| Up to $10^9 - 1$ | microseconds | us |
| Up to $10^{12} - 1$ | milliseconds | ms |
| Up to $60 \times 10^{12} - 1$ | seconds | s |
| Up to $3.6 \times 10^{15} - 1$ | minutes | min |
| Up to $8.64 \times 10^{16} - 1$ | hours | h |
| $8.64 \times 10^{16}$ and up | days | d |

```
mysql> SELECT FORMAT_PICO_TIME(3501), FORMAT_PICO_TIME(188732396662000);
+-----------------------+---------------------------------+
| FORMAT_PICO_TIME(3501) | FORMAT_PICO_TIME(188732396662000) |
+-----------------------+---------------------------------+
| 3.50 ns               | 3.15 min                        |
+-----------------------+---------------------------------+
```

`FORMAT_PICO_TIME()` was added in MySQL 8.0.16. It may be used instead of the `sys` schema `format_time()` function, keeping in mind these differences:

- To indicate minutes, `sys.format_time()` uses the `m` units indicator, whereas `FORMAT_PICO_TIME()` uses `min`.

- `sys.format_time()` uses the `w` (weeks) units indicator. `FORMAT_PICO_TIME()` does not.

- `PS_CURRENT_THREAD_ID()`

  Returns a `BIGINT UNSIGNED` value representing the Performance Schema thread ID assigned to the current connection.

  The thread ID return value is a value of the type given in the `THREAD_ID` column of Performance Schema tables.

  Performance Schema configuration affects `PS_CURRENT_THREAD_ID()` the same way as for `PS_THREAD_ID()`. For details, see the description of that function.

```
mysql> SELECT PS_CURRENT_THREAD_ID();
+------------------------+
| PS_CURRENT_THREAD_ID() |
+------------------------+
|                     52 |
+------------------------+
mysql> SELECT PS_THREAD_ID(CONNECTION_ID());
+-------------------------------+
| PS_THREAD_ID(CONNECTION_ID()) |
+-------------------------------+
|                            52 |
+-------------------------------+
```

  `PS_CURRENT_THREAD_ID()` was added in MySQL 8.0.16. It may be used as a shortcut for invoking the `sys` schema `ps_thread_id()` function with an argument of `NULL` or `CONNECTION_ID()`.

- `PS_THREAD_ID(`*`connection_id`*`)`

  Given a connection ID, returns a `BIGINT UNSIGNED` value representing the Performance Schema thread ID assigned to the connection ID, or `NULL` if no thread ID exists for the connection ID. The latter can occur for threads that are not instrumented, or if *`connection_id`* is `NULL`.

  The connection ID argument is a value of the type given in the `PROCESSLIST_ID` column of the Performance Schema `threads` table or the `Id` column of `SHOW PROCESSLIST` output.

  The thread ID return value is a value of the type given in the `THREAD_ID` column of Performance Schema tables.

  Performance Schema configuration affects `PS_THREAD_ID()` operation as follows. (These remarks also apply to `PS_CURRENT_THREAD_ID()`.)

  - Disabling the `thread_instrumentation` consumer disables statistics from being collected and aggregated at the thread level, but has no effect on `PS_THREAD_ID()`.
  - If `performance_schema_max_thread_instances` is not 0, the Performance Schema allocates memory for thread statistics and assigns an internal ID to each thread for which instance memory is available. If there are threads for which instance memory is not available, `PS_THREAD_ID()` returns `NULL`; in this case, `Performance_schema_thread_instances_lost` is nonzero.
  - If `performance_schema_max_thread_instances` is 0, the Performance Schema allocates no thread memory and `PS_THREAD_ID()` returns `NULL`.
  - If the Performance Schema itself is disabled, `PS_THREAD_ID()` produces an error.

```
mysql> SELECT PS_THREAD_ID(6);
+-----------------+
| PS_THREAD_ID(6) |
+-----------------+
|              45 |
+-----------------+
```

```
+-----------------+
```

`PS_THREAD_ID()` was added in MySQL 8.0.16. It may be used instead of the `sys` schema `ps_thread_id()` function, keeping in mind this difference:

- With an argument of `NULL`, `sys.ps_thread_id()` returns the thread ID for the current connection, whereas `PS_THREAD_ID()` returns `NULL`. To obtain the current connection thread ID, use `PS_CURRENT_THREAD_ID()` instead.

# 12.23 Internal Functions

**Table 12.28 Internal Functions**

| Name | Description | Introduced |
|---|---|---|
| `CAN_ACCESS_COLUMN()` | Internal use only | |
| `CAN_ACCESS_DATABASE()` | Internal use only | |
| `CAN_ACCESS_TABLE()` | Internal use only | |
| `CAN_ACCESS_USER()` | Internal use only | 8.0.22 |
| `CAN_ACCESS_VIEW()` | Internal use only | |
| `GET_DD_COLUMN_PRIVILEGES()` | Internal use only | |
| `GET_DD_CREATE_OPTIONS()` | Internal use only | |
| `GET_DD_INDEX_SUB_PART_LENGTH()` | Internal use only | |
| `INTERNAL_AUTO_INCREMENT()` | Internal use only | |
| `INTERNAL_AVG_ROW_LENGTH()` | Internal use only | |
| `INTERNAL_CHECK_TIME()` | Internal use only | |
| `INTERNAL_CHECKSUM()` | Internal use only | |
| `INTERNAL_DATA_FREE()` | Internal use only | |
| `INTERNAL_DATA_LENGTH()` | Internal use only | |
| `INTERNAL_DD_CHAR_LENGTH()` | Internal use only | |
| `INTERNAL_GET_COMMENT_OR_ERROR()` | Internal use only | |
| `INTERNAL_GET_ENABLED_ROLE_JSON()` | Internal use only | 8.0.19 |
| `INTERNAL_GET_HOSTNAME()` | Internal use only | 8.0.19 |
| `INTERNAL_GET_USERNAME()` | Internal use only | 8.0.19 |
| `INTERNAL_GET_VIEW_WARNING_OR_ERROR()` | Internal use only | |
| `INTERNAL_INDEX_COLUMN_CARDINALITY()` | Internal use only | |
| `INTERNAL_INDEX_LENGTH()` | Internal use only | |
| `INTERNAL_IS_ENABLED_ROLE()` | Internal use only | 8.0.19 |
| `INTERNAL_IS_MANDATORY_ROLE()` | Internal use only | 8.0.19 |
| `INTERNAL_KEYS_DISABLED()` | Internal use only | |
| `INTERNAL_MAX_DATA_LENGTH()` | Internal use only | |
| `INTERNAL_TABLE_ROWS()` | Internal use only | |
| `INTERNAL_UPDATE_TIME()` | Internal use only | |

The functions listed in this section are intended only for internal use by the server. Attempts by users to invoke them result in an error.

- `CAN_ACCESS_COLUMN(`*`ARGS`*`)`
- `CAN_ACCESS_DATABASE(`*`ARGS`*`)`

- CAN_ACCESS_TABLE(*ARGS*)
- CAN_ACCESS_USER(*ARGS*)
- CAN_ACCESS_VIEW(*ARGS*)
- GET_DD_COLUMN_PRIVILEGES(*ARGS*)
- GET_DD_CREATE_OPTIONS(*ARGS*)
- GET_DD_INDEX_SUB_PART_LENGTH(*ARGS*)
- INTERNAL_AUTO_INCREMENT(*ARGS*)
- INTERNAL_AVG_ROW_LENGTH(*ARGS*)
- INTERNAL_CHECK_TIME(*ARGS*)
- INTERNAL_CHECKSUM(*ARGS*)
- INTERNAL_DATA_FREE(*ARGS*)
- INTERNAL_DATA_LENGTH(*ARGS*)
- INTERNAL_DD_CHAR_LENGTH(*ARGS*)
- INTERNAL_GET_COMMENT_OR_ERROR(*ARGS*)
- INTERNAL_GET_ENABLED_ROLE_JSON(*ARGS*)
- INTERNAL_GET_HOSTNAME(*ARGS*)
- INTERNAL_GET_USERNAME(*ARGS*)
- INTERNAL_GET_VIEW_WARNING_OR_ERROR(*ARGS*)
- INTERNAL_INDEX_COLUMN_CARDINALITY(*ARGS*)
- INTERNAL_INDEX_LENGTH(*ARGS*)
- INTERNAL_IS_ENABLED_ROLE(*ARGS*)
- INTERNAL_IS_MANDATORY_ROLE(*ARGS*)
- INTERNAL_KEYS_DISABLED(*ARGS*)
- INTERNAL_MAX_DATA_LENGTH(*ARGS*)
- INTERNAL_TABLE_ROWS(*ARGS*)
- INTERNAL_UPDATE_TIME(*ARGS*)
- IS_VISIBLE_DD_OBJECT(*ARGS*)

## 12.24 Miscellaneous Functions

**Table 12.29 Miscellaneous Functions**

| Name | Description | Introduced | Deprecated |
|---|---|---|---|
| ANY_VALUE() | Suppress ONLY_FULL_GROUP_BY value rejection | | |
| BIN_TO_UUID() | Convert binary UUID to string | | |
| DEFAULT() | Return the default value for a table column | | |

| Name | Description | Introduced | Deprecated |
|---|---|---|---|
| GROUPING() | Distinguish super-aggregate ROLLUP rows from regular rows | | |
| INET_ATON() | Return the numeric value of an IP address | | |
| INET_NTOA() | Return the IP address from a numeric value | | |
| INET6_ATON() | Return the numeric value of an IPv6 address | | |
| INET6_NTOA() | Return the IPv6 address from a numeric value | | |
| IS_IPV4() | Whether argument is an IPv4 address | | |
| IS_IPV4_COMPAT() | Whether argument is an IPv4-compatible address | | |
| IS_IPV4_MAPPED() | Whether argument is an IPv4-mapped address | | |
| IS_IPV6() | Whether argument is an IPv6 address | | |
| IS_UUID() | Whether argument is a valid UUID | | |
| MASTER_POS_WAIT() | Block until the replica has read and applied all updates up to the specified position | | 8.0.26 |
| NAME_CONST() | Cause the column to have the given name | | |
| SLEEP() | Sleep for a number of seconds | | |
| SOURCE_POS_WAIT() | Block until the replica has read and applied all updates up to the specified position | 8.0.26 | |
| UUID() | Return a Universal Unique Identifier (UUID) | | |
| UUID_SHORT() | Return an integer-valued universal identifier | | |
| UUID_TO_BIN() | Convert string UUID to binary | | |
| VALUES() | Define the values to be used during an INSERT | | |

- ANY_VALUE(*arg*)

  This function is useful for GROUP BY queries when the ONLY_FULL_GROUP_BY SQL mode is enabled, for cases when MySQL rejects a query that you know is valid for reasons that MySQL

cannot determine. The function return value and type are the same as the return value and type of its argument, but the function result is not checked for the ONLY_FULL_GROUP_BY SQL mode.

For example, if name is a nonindexed column, the following query fails with ONLY_FULL_GROUP_BY enabled:

```
mysql> SELECT name, address, MAX(age) FROM t GROUP BY name;
ERROR 1055 (42000): Expression #2 of SELECT list is not in GROUP
BY clause and contains nonaggregated column 'mydb.t.address' which
is not functionally dependent on columns in GROUP BY clause; this
is incompatible with sql_mode=only_full_group_by
```

The failure occurs because address is a nonaggregated column that is neither named among GROUP BY columns nor functionally dependent on them. As a result, the address value for rows within each name group is nondeterministic. There are multiple ways to cause MySQL to accept the query:

- Alter the table to make name a primary key or a unique NOT NULL column. This enables MySQL to determine that address is functionally dependent on name; that is, address is uniquely determined by name. (This technique is inapplicable if NULL must be permitted as a valid name value.)
- Use ANY_VALUE() to refer to address:

  ```
  SELECT name, ANY_VALUE(address), MAX(age) FROM t GROUP BY name;
  ```

  In this case, MySQL ignores the nondeterminism of address values within each name group and accepts the query. This may be useful if you simply do not care which value of a nonaggregated column is chosen for each group. ANY_VALUE() is not an aggregate function, unlike functions such as SUM() or COUNT(). It simply acts to suppress the test for nondeterminism.
- Disable ONLY_FULL_GROUP_BY. This is equivalent to using ANY_VALUE() with ONLY_FULL_GROUP_BY enabled, as described in the previous item.

ANY_VALUE() is also useful if functional dependence exists between columns but MySQL cannot determine it. The following query is valid because age is functionally dependent on the grouping column age-1, but MySQL cannot tell that and rejects the query with ONLY_FULL_GROUP_BY enabled:

```
SELECT age FROM t GROUP BY age-1;
```

To cause MySQL to accept the query, use ANY_VALUE():

```
SELECT ANY_VALUE(age) FROM t GROUP BY age-1;
```

ANY_VALUE() can be used for queries that refer to aggregate functions in the absence of a GROUP BY clause:

```
mysql> SELECT name, MAX(age) FROM t;
ERROR 1140 (42000): In aggregated query without GROUP BY, expression
#1 of SELECT list contains nonaggregated column 'mydb.t.name'; this
is incompatible with sql_mode=only_full_group_by
```

Without GROUP BY, there is a single group and it is nondeterministic which name value to choose for the group. ANY_VALUE() tells MySQL to accept the query:

```
SELECT ANY_VALUE(name), MAX(age) FROM t;
```

It may be that, due to some property of a given data set, you know that a selected nonaggregated column is effectively functionally dependent on a GROUP BY column. For example, an application

may enforce uniqueness of one column with respect to another. In this case, using `ANY_VALUE()` for the effectively functionally dependent column may make sense.

For additional discussion, see Section 12.20.3, “MySQL Handling of GROUP BY”.

- `BIN_TO_UUID(binary_uuid)`, `BIN_TO_UUID(binary_uuid, swap_flag)`

  `BIN_TO_UUID()` is the inverse of `UUID_TO_BIN()`. It converts a binary UUID to a string UUID and returns the result. The binary value should be a UUID as a `VARBINARY(16)` value. The return value is a string of five hexadecimal numbers separated by dashes. (For details about this format, see the `UUID()` function description.) If the UUID argument is `NULL`, the return value is `NULL`. If any argument is invalid, an error occurs.

  `BIN_TO_UUID()` takes one or two arguments:

  - The one-argument form takes a binary UUID value. The UUID value is assumed not to have its time-low and time-high parts swapped. The string result is in the same order as the binary argument.
  - The two-argument form takes a binary UUID value and a swap-flag value:
    - If *`swap_flag`* is 0, the two-argument form is equivalent to the one-argument form. The string result is in the same order as the binary argument.
    - If *`swap_flag`* is 1, the UUID value is assumed to have its time-low and time-high parts swapped. These parts are swapped back to their original position in the result value.

  For usage examples and information about time-part swapping, see the `UUID_TO_BIN()` function description.

- `DEFAULT(col_name)`

  Returns the default value for a table column. An error results if the column has no default value.

  The use of `DEFAULT(col_name)` to specify the default value for a named column is permitted only for columns that have a literal default value, not for columns that have an expression default value.

  ```
  mysql> UPDATE t SET i = DEFAULT(i)+1 WHERE id < 100;
  ```

- `FORMAT(X,D)`

  Formats the number *`X`* to a format like `'#,###,###.##'`, rounded to *`D`* decimal places, and returns the result as a string. For details, see Section 12.8, “String Functions and Operators”.

- `GROUPING(expr [, expr] ...)`

  For `GROUP BY` queries that include a `WITH ROLLUP` modifier, the `ROLLUP` operation produces super-aggregate output rows where `NULL` represents the set of all values. The `GROUPING()` function enables you to distinguish `NULL` values for super-aggregate rows from `NULL` values in regular grouped rows.

  `GROUPING()` is permitted in the select list, `HAVING` clause, and (as of MySQL 8.0.12) `ORDER BY` clause.

  Each argument to `GROUPING()` must be an expression that exactly matches an expression in the `GROUP BY` clause. The expression cannot be a positional specifier. For each expression, `GROUPING()` produces 1 if the expression value in the current row is a `NULL` representing a super-aggregate value. Otherwise, `GROUPING()` produces 0, indicating that the expression value is a `NULL` for a regular result row or is not `NULL`.

  Suppose that table `t1` contains these rows, where `NULL` indicates something like “other” or “unknown”:

```
mysql> SELECT * FROM t1;
+------+-------+----------+
| name | size  | quantity |
+------+-------+----------+
| ball | small |       10 |
| ball | large |       20 |
| ball | NULL  |        5 |
| hoop | small |       15 |
| hoop | large |        5 |
| hoop | NULL  |        3 |
+------+-------+----------+
```

A summary of the table without `WITH ROLLUP` looks like this:

```
mysql> SELECT name, size, SUM(quantity) AS quantity
       FROM t1
       GROUP BY name, size;
+------+-------+----------+
| name | size  | quantity |
+------+-------+----------+
| ball | small |       10 |
| ball | large |       20 |
| ball | NULL  |        5 |
| hoop | small |       15 |
| hoop | large |        5 |
| hoop | NULL  |        3 |
+------+-------+----------+
```

The result contains `NULL` values, but those do not represent super-aggregate rows because the query does not include `WITH ROLLUP`.

Adding `WITH ROLLUP` produces super-aggregate summary rows containing additional `NULL` values. However, without comparing this result to the previous one, it is not easy to see which `NULL` values occur in super-aggregate rows and which occur in regular grouped rows:

```
mysql> SELECT name, size, SUM(quantity) AS quantity
       FROM t1
       GROUP BY name, size WITH ROLLUP;
+------+-------+----------+
| name | size  | quantity |
+------+-------+----------+
| ball | NULL  |        5 |
| ball | large |       20 |
| ball | small |       10 |
| ball | NULL  |       35 |
| hoop | NULL  |        3 |
| hoop | large |        5 |
| hoop | small |       15 |
| hoop | NULL  |       23 |
| NULL | NULL  |       58 |
+------+-------+----------+
```

To distinguish `NULL` values in super-aggregate rows from those in regular grouped rows, use `GROUPING()`, which returns 1 only for super-aggregate `NULL` values:

```
mysql> SELECT
         name, size, SUM(quantity) AS quantity,
         GROUPING(name) AS grp_name,
         GROUPING(size) AS grp_size
       FROM t1
       GROUP BY name, size WITH ROLLUP;
+------+-------+----------+----------+----------+
| name | size  | quantity | grp_name | grp_size |
+------+-------+----------+----------+----------+
| ball | NULL  |        5 |        0 |        0 |
| ball | large |       20 |        0 |        0 |
| ball | small |       10 |        0 |        0 |
| ball | NULL  |       35 |        0 |        1 |
| hoop | NULL  |        3 |        0 |        0 |
```

```
| hoop | large |        5 |        0 |        0 |
| hoop | small |       15 |        0 |        0 |
| hoop | NULL  |       23 |        0 |        1 |
| NULL | NULL  |       58 |        1 |        1 |
+------+-------+----------+----------+----------+
```

Common uses for `GROUPING()`:

- Substitute a label for super-aggregate `NULL` values:

```
mysql> SELECT
         IF(GROUPING(name) = 1, 'All items', name) AS name,
         IF(GROUPING(size) = 1, 'All sizes', size) AS size,
         SUM(quantity) AS quantity
       FROM t1
       GROUP BY name, size WITH ROLLUP;
+-----------+-----------+----------+
| name      | size      | quantity |
+-----------+-----------+----------+
| ball      | NULL      |        5 |
| ball      | large     |       20 |
| ball      | small     |       10 |
| ball      | All sizes |       35 |
| hoop      | NULL      |        3 |
| hoop      | large     |        5 |
| hoop      | small     |       15 |
| hoop      | All sizes |       23 |
| All items | All sizes |       58 |
+-----------+-----------+----------+
```

- Return only super-aggregate lines by filtering out the regular grouped lines:

```
mysql> SELECT name, size, SUM(quantity) AS quantity
       FROM t1
       GROUP BY name, size WITH ROLLUP
       HAVING GROUPING(name) = 1 OR GROUPING(size) = 1;
+------+------+----------+
| name | size | quantity |
+------+------+----------+
| ball | NULL |       35 |
| hoop | NULL |       23 |
| NULL | NULL |       58 |
+------+------+----------+
```

`GROUPING()` permits multiple expression arguments. In this case, the `GROUPING()` return value represents a bitmask combined from the results for each expression, where the lowest-order bit corresponds to the result for the rightmost expression. For example, with three expression arguments, `GROUPING(`*`expr1, expr2, expr3`*`)` is evaluated like this:

```
  result for GROUPING(expr3)
+ result for GROUPING(expr2) << 1
+ result for GROUPING(expr1) << 2
```

The following query shows how `GROUPING()` results for single arguments combine for a multiple-argument call to produce a bitmask value:

```
mysql> SELECT
         name, size, SUM(quantity) AS quantity,
         GROUPING(name) AS grp_name,
         GROUPING(size) AS grp_size,
       GROUPING(name, size) AS grp_all
       FROM t1
       GROUP BY name, size WITH ROLLUP;
+------+-------+----------+----------+----------+---------+
| name | size  | quantity | grp_name | grp_size | grp_all |
+------+-------+----------+----------+----------+---------+
| ball | NULL  |        5 |        0 |        0 |       0 |
| ball | large |       20 |        0 |        0 |       0 |
| ball | small |       10 |        0 |        0 |       0 |
| ball | NULL  |       35 |        0 |        1 |       1 |
```

```
| hoop | NULL  |        3 |        0 |        0 |       0 |
| hoop | large |        5 |        0 |        0 |       0 |
| hoop | small |       15 |        0 |        0 |       0 |
| hoop | NULL  |       23 |        0 |        1 |       1 |
| NULL | NULL  |       58 |        1 |        1 |       3 |
+------+-------+----------+----------+----------+---------+
```

With multiple expression arguments, the `GROUPING()` return value is nonzero if any expression represents a super-aggregate value. Multiple-argument `GROUPING()` syntax thus provides a simpler way to write the earlier query that returned only super-aggregate rows, by using a single multiple-argument `GROUPING()` call rather than multiple single-argument calls:

```
mysql> SELECT name, size, SUM(quantity) AS quantity
       FROM t1
       GROUP BY name, size WITH ROLLUP
       HAVING GROUPING(name, size) <> 0;
+------+------+----------+
| name | size | quantity |
+------+------+----------+
| ball | NULL |       35 |
| hoop | NULL |       23 |
| NULL | NULL |       58 |
+------+------+----------+
```

Use of `GROUPING()` is subject to these limitations:

- Do not use subquery `GROUP BY` expressions as `GROUPING()` arguments because matching might fail. For example, matching fails for this query:

  ```
  mysql> SELECT GROUPING((SELECT MAX(name) FROM t1))
         FROM t1
         GROUP BY (SELECT MAX(name) FROM t1) WITH ROLLUP;
  ERROR 3580 (HY000): Argument #1 of GROUPING function is not in GROUP BY
  ```

- `GROUP BY` literal expressions should not be used within a `HAVING` clause as `GROUPING()` arguments. Due to differences between when the optimizer evaluates `GROUP BY` and `HAVING`, matching may succeed but `GROUPING()` evaluation does not produce the expected result. Consider this query:

  ```
  SELECT a AS f1, 'w' AS f2
  FROM t
  GROUP BY f1, f2 WITH ROLLUP
  HAVING GROUPING(f2) = 1;
  ```

  `GROUPING()` is evaluated earlier for the literal constant expression than for the `HAVING` clause as a whole and returns 0. To check whether a query such as this is affected, use `EXPLAIN` and look for `Impossible having` in the `Extra` column.

For more information about `WITH ROLLUP` and `GROUPING()`, see Section 12.20.2, “GROUP BY Modifiers”.

- `INET_ATON(`*`expr`*`)`

  Given the dotted-quad representation of an IPv4 network address as a string, returns an integer that represents the numeric value of the address in network byte order (big endian). `INET_ATON()` returns `NULL` if it does not understand its argument, or if *`expr`* is `NULL`.

  ```
  mysql> SELECT INET_ATON('10.0.5.9');
  ```

```
        -> 167773449
```

For this example, the return value is calculated as $10\times256^3 + 0\times256^2 + 5\times256 + 9$.

`INET_ATON()` may or may not return a non-`NULL` result for short-form IP addresses (such as `'127.1'` as a representation of `'127.0.0.1'`). Because of this, `INET_ATON()`a should not be used for such addresses.

**Note**

To store values generated by `INET_ATON()`, use an `INT UNSIGNED` column rather than `INT`, which is signed. If you use a signed column, values corresponding to IP addresses for which the first octet is greater than 127 cannot be stored correctly. See Section 11.1.7, “Out-of-Range and Overflow Handling”.

- `INET_NTOA(`*`expr`*`)`

  Given a numeric IPv4 network address in network byte order, returns the dotted-quad string representation of the address as a string in the connection character set. `INET_NTOA()` returns `NULL` if it does not understand its argument.

```
mysql> SELECT INET_NTOA(167773449);
        -> '10.0.5.9'
```

- `INET6_ATON(`*`expr`*`)`

  Given an IPv6 or IPv4 network address as a string, returns a binary string that represents the numeric value of the address in network byte order (big endian). Because numeric-format IPv6 addresses require more bytes than the largest integer type, the representation returned by this function has the `VARBINARY` data type: `VARBINARY(16)` for IPv6 addresses and `VARBINARY(4)` for IPv4 addresses. If the argument is not a valid address, or if it is `NULL`, `INET6_ATON()` returns `NULL`.

  The following examples use `HEX()` to display the `INET6_ATON()` result in printable form:

```
mysql> SELECT HEX(INET6_ATON('fdfe::5a55:caff:fefa:9089'));
        -> 'FDFE0000000000005A55CAFFFEFA9089'
mysql> SELECT HEX(INET6_ATON('10.0.5.9'));
        -> '0A000509'
```

  `INET6_ATON()` observes several constraints on valid arguments. These are given in the following list along with examples.

  - A trailing zone ID is not permitted, as in `fe80::3%1` or `fe80::3%eth0`.
  - A trailing network mask is not permitted, as in `2001:45f:3:ba::/64` or `198.51.100.0/24`.
  - For values representing IPv4 addresses, only classless addresses are supported. Classful addresses such as `198.51.1` are rejected. A trailing port number is not permitted, as in `198.51.100.2:8080`. Hexadecimal numbers in address components are not permitted, as in `198.0xa0.1.2`. Octal numbers are not supported: `198.51.010.1` is treated as `198.51.10.1`, not `198.51.8.1`. These IPv4 constraints also apply to IPv6 addresses that have IPv4 address parts, such as IPv4-compatible or IPv4-mapped addresses.

  To convert an IPv4 address *`expr`* represented in numeric form as an `INT` value to an IPv6 address represented in numeric form as a `VARBINARY` value, use this expression:

```
INET6_ATON(INET_NTOA(expr))
```

  For example:

```
mysql> SELECT HEX(INET6_ATON(INET_NTOA(167773449)));
```

```
        -> '0A000509'
```

If `INET6_ATON()` is invoked from within the `mysql` client, binary strings display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

- `INET6_NTOA(expr)`

  Given an IPv6 or IPv4 network address represented in numeric form as a binary string, returns the string representation of the address as a string in the connection character set. If the argument is not a valid address, or if it is `NULL`, `INET6_NTOA()` returns `NULL`.

  `INET6_NTOA()` has these properties:

  - It does not use operating system functions to perform conversions, thus the output string is platform independent.
  - The return string has a maximum length of 39 (4 x 8 + 7). Given this statement:

    ```
    CREATE TABLE t AS SELECT INET6_NTOA(expr) AS c1;
    ```

    The resulting table would have this definition:

    ```
    CREATE TABLE t (c1 VARCHAR(39) CHARACTER SET utf8mb3 DEFAULT NULL);
    ```

  - The return string uses lowercase letters for IPv6 addresses.

```
mysql> SELECT INET6_NTOA(INET6_ATON('fdfe::5a55:caff:fefa:9089'));
        -> 'fdfe::5a55:caff:fefa:9089'
mysql> SELECT INET6_NTOA(INET6_ATON('10.0.5.9'));
        -> '10.0.5.9'

mysql> SELECT INET6_NTOA(UNHEX('FDFE0000000000005A55CAFFFEFA9089'));
        -> 'fdfe::5a55:caff:fefa:9089'
mysql> SELECT INET6_NTOA(UNHEX('0A000509'));
        -> '10.0.5.9'
```

If `INET6_NTOA()` is invoked from within the `mysql` client, binary strings display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

- `IS_IPV4(expr)`

  Returns 1 if the argument is a valid IPv4 address specified as a string, 0 otherwise. Returns `NULL` if *`expr`* is `NULL`.

```
mysql> SELECT IS_IPV4('10.0.5.9'), IS_IPV4('10.0.5.256');
        -> 1, 0
```

  For a given argument, if `IS_IPV4()` returns 1, `INET_ATON()` (and `INET6_ATON()`) returns non-`NULL`. The converse statement is not true: In some cases, `INET_ATON()` returns non-`NULL` when `IS_IPV4()` returns 0.

  As implied by the preceding remarks, `IS_IPV4()` is more strict than `INET_ATON()` about what constitutes a valid IPv4 address, so it may be useful for applications that need to perform strong checks against invalid values. Alternatively, use `INET6_ATON()` to convert IPv4 addresses to internal form and check for a `NULL` result (which indicates an invalid address). `INET6_ATON()` is equally strong as `IS_IPV4()` about checking IPv4 addresses.

- `IS_IPV4_COMPAT(expr)`

  This function takes an IPv6 address represented in numeric form as a binary string, as returned by `INET6_ATON()`. It returns 1 if the argument is a valid IPv4-compatible IPv6 address, 0 otherwise

(unless *expr* is NULL, in which case the function returns NULL). IPv4-compatible addresses have the form ::*ipv4_address*.

```
mysql> SELECT IS_IPV4_COMPAT(INET6_ATON('::10.0.5.9'));
        -> 1
mysql> SELECT IS_IPV4_COMPAT(INET6_ATON('::ffff:10.0.5.9'));
        -> 0
```

The IPv4 part of an IPv4-compatible address can also be represented using hexadecimal notation. For example, 198.51.100.1 has this raw hexadecimal value:

```
mysql> SELECT HEX(INET6_ATON('198.51.100.1'));
        -> 'C6336401'
```

Expressed in IPv4-compatible form, ::198.51.100.1 is equivalent to ::c0a8:0001 or (without leading zeros) ::c0a8:1

```
mysql> SELECT
    ->   IS_IPV4_COMPAT(INET6_ATON('::198.51.100.1')),
    ->   IS_IPV4_COMPAT(INET6_ATON('::c0a8:0001')),
    ->   IS_IPV4_COMPAT(INET6_ATON('::c0a8:1'));
        -> 1, 1, 1
```

- IS_IPV4_MAPPED(*expr*)

  This function takes an IPv6 address represented in numeric form as a binary string, as returned by INET6_ATON(). It returns 1 if the argument is a valid IPv4-mapped IPv6 address, 0 otherwise, unless *expr* is NULL, in which case the function returns NULL. IPv4-mapped addresses have the form ::ffff:*ipv4_address*.

  ```
  mysql> SELECT IS_IPV4_MAPPED(INET6_ATON('::10.0.5.9'));
          -> 0
  mysql> SELECT IS_IPV4_MAPPED(INET6_ATON('::ffff:10.0.5.9'));
          -> 1
  ```

  As with IS_IPV4_COMPAT() the IPv4 part of an IPv4-mapped address can also be represented using hexadecimal notation:

  ```
  mysql> SELECT
      ->   IS_IPV4_MAPPED(INET6_ATON('::ffff:198.51.100.1')),
      ->   IS_IPV4_MAPPED(INET6_ATON('::ffff:c0a8:0001')),
      ->   IS_IPV4_MAPPED(INET6_ATON('::ffff:c0a8:1'));
          -> 1, 1, 1
  ```

- IS_IPV6(*expr*)

  Returns 1 if the argument is a valid IPv6 address specified as a string, 0 otherwise, unless *expr* is NULL, in which case the function returns NULL. This function does not consider IPv4 addresses to be valid IPv6 addresses.

  ```
  mysql> SELECT IS_IPV6('10.0.5.9'), IS_IPV6('::1');
          -> 0, 1
  ```

  For a given argument, if IS_IPV6() returns 1, INET6_ATON() returns non-NULL.

- `IS_UUID(string_uuid)`

  Returns 1 if the argument is a valid string-format UUID, 0 if the argument is not a valid UUID, and `NULL` if the argument is `NULL`.

  “Valid” means that the value is in a format that can be parsed. That is, it has the correct length and contains only the permitted characters (hexadecimal digits in any lettercase and, optionally, dashes and curly braces). This format is most common:

  ```
  aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee
  ```

  These other formats are also permitted:

  ```
  aaaaaaaabbbbccccddddeeeeeeeeeeee
  {aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee}
  ```

  For the meanings of fields within the value, see the `UUID()` function description.

  ```
  mysql> SELECT IS_UUID('6ccd780c-baba-1026-9564-5b8c656024db');
  +-----------------------------------------------+
  | IS_UUID('6ccd780c-baba-1026-9564-5b8c656024db') |
  +-----------------------------------------------+
  |                                             1 |
  +-----------------------------------------------+
  mysql> SELECT IS_UUID('6CCD780C-BABA-1026-9564-5B8C656024DB');
  +-----------------------------------------------+
  | IS_UUID('6CCD780C-BABA-1026-9564-5B8C656024DB') |
  +-----------------------------------------------+
  |                                             1 |
  +-----------------------------------------------+
  mysql> SELECT IS_UUID('6ccd780cbaba102695645b8c656024db');
  +-------------------------------------------+
  | IS_UUID('6ccd780cbaba102695645b8c656024db') |
  +-------------------------------------------+
  |                                         1 |
  +-------------------------------------------+
  mysql> SELECT IS_UUID('{6ccd780c-baba-1026-9564-5b8c656024db}');
  +-------------------------------------------------+
  | IS_UUID('{6ccd780c-baba-1026-9564-5b8c656024db}') |
  +-------------------------------------------------+
  |                                               1 |
  +-------------------------------------------------+
  mysql> SELECT IS_UUID('6ccd780c-baba-1026-9564-5b8c6560');
  +-------------------------------------------+
  | IS_UUID('6ccd780c-baba-1026-9564-5b8c6560') |
  +-------------------------------------------+
  |                                         0 |
  +-------------------------------------------+
  mysql> SELECT IS_UUID(RAND());
  +-----------------+
  | IS_UUID(RAND()) |
  +-----------------+
  |               0 |
  +-----------------+
  ```

- `MASTER_POS_WAIT(log_name,log_pos[,timeout][,channel])`

  This function is for control of source/replica synchronization. It blocks until the replica has read and applied all updates up to the specified position in the source's binary log. From MySQL 8.0.26, `MASTER_POS_WAIT()` is deprecated and the alias `SOURCE_POS_WAIT()` should be used instead. In releases before MySQL 8.0.26, use `MASTER_POS_WAIT()`.

  The return value is the number of log events the replica had to wait for to advance to the specified position. The function returns `NULL` if the replication SQL thread is not started, the replica's source information is not initialized, the arguments are incorrect, or an error occurs. It returns `-1` if the timeout has been exceeded. If the replication SQL thread stops while `MASTER_POS_WAIT()` is

waiting, the function returns `NULL`. If the replica is past the specified position, the function returns immediately.

If the binary log file position has been marked as invalid, the function waits until a valid file position is known. The binary log file position can be marked as invalid when the `CHANGE REPLICATION SOURCE TO` option `GTID_ONLY` is set for the replication channel, and the server is restarted or replication is stopped. The file position becomes valid after a transaction is successfully applied past the given file position. If the applier does not reach the stated position, the function waits until the timeout. Use a `SHOW REPLICA STATUS` statement to check if the binary log file position has been marked as invalid.

On a multithreaded replica, the function waits until expiry of the limit set by the `replica_checkpoint_group`, `slave_checkpoint_group`, `replica_checkpoint_period` or `slave_checkpoint_period` system variable, when the checkpoint operation is called to update the status of the replica. Depending on the setting for the system variables, the function might therefore return some time after the specified position was reached.

If binary log transaction compression is in use and the transaction payload at the specified position is compressed (as a `Transaction_payload_event`), the function waits until the whole transaction has been read and applied, and the positions have updated.

If a *`timeout`* value is specified, `MASTER_POS_WAIT()` stops waiting when *`timeout`* seconds have elapsed. *`timeout`* must be greater than 0; a zero or negative *`timeout`* means no timeout.

The optional *`channel`* value enables you to name which replication channel the function applies to. See Section 17.2.2, “Replication Channels” for more information.

This function is unsafe for statement-based replication. A warning is logged if you use this function when `binlog_format` is set to `STATEMENT`.

- `NAME_CONST(`*`name`*`,`*`value`*`)`

  Returns the given value. When used to produce a result set column, `NAME_CONST()` causes the column to have the given name. The arguments should be constants.

```
mysql> SELECT NAME_CONST('myname', 14);
+--------+
| myname |
+--------+
|     14 |
+--------+
```

  This function is for internal use only. The server uses it when writing statements from stored programs that contain references to local program variables, as described in Section 25.7, “Stored Program Binary Logging”. You might see this function in the output from `mysqlbinlog`.

  For your applications, you can obtain exactly the same result as in the example just shown by using simple aliasing, like this:

```
mysql> SELECT 14 AS myname;
+--------+
| myname |
+--------+
|     14 |
+--------+
1 row in set (0.00 sec)
```

  See Section 13.2.13, “SELECT Statement”, for more information about column aliases.

- `SLEEP(duration)`

  Sleeps (pauses) for the number of seconds given by the *`duration`* argument, then returns 0. The duration may have a fractional part. If the argument is `NULL` or negative, `SLEEP()` produces a warning, or an error in strict SQL mode.

  When sleep returns normally (without interruption), it returns 0:

```
mysql> SELECT SLEEP(1000);
+-------------+
| SLEEP(1000) |
+-------------+
|           0 |
+-------------+
```

  When `SLEEP()` is the only thing invoked by a query that is interrupted, it returns 1 and the query itself returns no error. This is true whether the query is killed or times out:

  - This statement is interrupted using `KILL QUERY` from another session:

```
mysql> SELECT SLEEP(1000);
+-------------+
| SLEEP(1000) |
+-------------+
|           1 |
+-------------+
```

  - This statement is interrupted by timing out:

```
mysql> SELECT /*+ MAX_EXECUTION_TIME(1) */ SLEEP(1000);
+-------------+
| SLEEP(1000) |
+-------------+
|           1 |
+-------------+
```

  When `SLEEP()` is only part of a query that is interrupted, the query returns an error:

  - This statement is interrupted using `KILL QUERY` from another session:

```
mysql> SELECT 1 FROM t1 WHERE SLEEP(1000);
ERROR 1317 (70100): Query execution was interrupted
```

  - This statement is interrupted by timing out:

```
mysql> SELECT /*+ MAX_EXECUTION_TIME(1000) */ 1 FROM t1 WHERE SLEEP(1000);
ERROR 3024 (HY000): Query execution was interrupted, maximum statement
execution time exceeded
```

  This function is unsafe for statement-based replication. A warning is logged if you use this function when `binlog_format` is set to `STATEMENT`.

- `SOURCE_POS_WAIT(log_name,log_pos[,timeout][,channel])`

  This function is for control of source/replica synchronization. It blocks until the replica has read and applied all updates up to the specified position in the source's binary log. From MySQL 8.0.26, use `SOURCE_POS_WAIT()` in place of `MASTER_POS_WAIT()`, which is deprecated from that release. In releases before MySQL 8.0.26, use `MASTER_POS_WAIT()`.

  The return value is the number of log events the replica had to wait for to advance to the specified position. The function returns `NULL` if the replication SQL thread is not started, the replica's source information is not initialized, the arguments are incorrect, or an error occurs. It returns `-1` if the timeout has been exceeded. If the replication SQL thread stops while `SOURCE_POS_WAIT()` is

waiting, the function returns `NULL`. If the replica is past the specified position, the function returns immediately.

If the binary log file position has been marked as invalid, the function waits until a valid file position is known. The binary log file position can be marked as invalid when the `CHANGE REPLICATION SOURCE TO` option `GTID_ONLY` is set for the replication channel, and the server is restarted or replication is stopped. The file position becomes valid after a transaction is successfully applied past the given file position. If the applier does not reach the stated position, the function waits until the timeout. Use a `SHOW REPLICA STATUS` statement to check if the binary log file position has been marked as invalid.

On a multithreaded replica, the function waits until expiry of the limit set by the `replica_checkpoint_group` or `replica_checkpoint_period` system variable, when the checkpoint operation is called to update the status of the replica. Depending on the setting for the system variables, the function might therefore return some time after the specified position was reached.

If binary log transaction compression is in use and the transaction payload at the specified position is compressed (as a `Transaction_payload_event`), the function waits until the whole transaction has been read and applied, and the positions have updated.

If a *`timeout`* value is specified, `SOURCE_POS_WAIT()` stops waiting when *`timeout`* seconds have elapsed. *`timeout`* must be greater than 0; a zero or negative *`timeout`* means no timeout.

The optional *`channel`* value enables you to name which replication channel the function applies to. See Section 17.2.2, “Replication Channels” for more information.

This function is unsafe for statement-based replication. A warning is logged if you use this function when `binlog_format` is set to `STATEMENT`.

- `UUID()`

  Returns a Universal Unique Identifier (UUID) generated according to RFC 4122, “A Universally Unique IDentifier (UUID) URN Namespace” (http://www.ietf.org/rfc/rfc4122.txt).

  A UUID is designed as a number that is globally unique in space and time. Two calls to `UUID()` are expected to generate two different values, even if these calls are performed on two separate devices not connected to each other.

  

  **Warning**

  Although `UUID()` values are intended to be unique, they are not necessarily unguessable or unpredictable. If unpredictability is required, UUID values should be generated some other way.

  `UUID()` returns a value that conforms to UUID version 1 as described in RFC 4122. The value is a 128-bit number represented as a `utf8mb3` string of five hexadecimal numbers in `aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee` format:

  - The first three numbers are generated from the low, middle, and high parts of a timestamp. The high part also includes the UUID version number.
  - The fourth number preserves temporal uniqueness in case the timestamp value loses monotonicity (for example, due to daylight saving time).
  - The fifth number is an IEEE 802 node number that provides spatial uniqueness. A random number is substituted if the latter is not available (for example, because the host device has no Ethernet card, or it is unknown how to find the hardware address of an interface on the host operating

system). In this case, spatial uniqueness cannot be guaranteed. Nevertheless, a collision should have *very* low probability.

The MAC address of an interface is taken into account only on FreeBSD, Linux, and Windows. On other operating systems, MySQL uses a randomly generated 48-bit number.

```
mysql> SELECT UUID();
        -> '6ccd780c-baba-1026-9564-5b8c656024db'
```

To convert between string and binary UUID values, use the `UUID_TO_BIN()` and `BIN_TO_UUID()` functions. To check whether a string is a valid UUID value, use the `IS_UUID()` function.

This function is unsafe for statement-based replication. A warning is logged if you use this function when `binlog_format` is set to `STATEMENT`.

- `UUID_SHORT()`

Returns a “short” universal identifier as a 64-bit unsigned integer. Values returned by `UUID_SHORT()` differ from the string-format 128-bit identifiers returned by the `UUID()` function and have different uniqueness properties. The value of `UUID_SHORT()` is guaranteed to be unique if the following conditions hold:

- The `server_id` value of the current server is between 0 and 255 and is unique among your set of source and replica servers
- You do not set back the system time for your server host between `mysqld` restarts
- You invoke `UUID_SHORT()` on average fewer than 16 million times per second between `mysqld` restarts

The `UUID_SHORT()` return value is constructed this way:

```
  (server_id & 255) << 56
+ (server_startup_time_in_seconds << 24)
+ incremented_variable++;
```

```
mysql> SELECT UUID_SHORT();
        -> 92395783831158784
```

**Note**

`UUID_SHORT()` does not work with statement-based replication.

- `UUID_TO_BIN(`*`string_uuid`*`)`, `UUID_TO_BIN(`*`string_uuid, swap_flag`*`)`

Converts a string UUID to a binary UUID and returns the result. (The `IS_UUID()` function description lists the permitted string UUID formats.) The return binary UUID is a `VARBINARY(16)` value. If the UUID argument is `NULL`, the return value is `NULL`. If any argument is invalid, an error occurs.

`UUID_TO_BIN()` takes one or two arguments:

- The one-argument form takes a string UUID value. The binary result is in the same order as the string argument.
- The two-argument form takes a string UUID value and a flag value:
  - If *`swap_flag`* is 0, the two-argument form is equivalent to the one-argument form. The binary result is in the same order as the string argument.
  - If *`swap_flag`* is 1, the format of the return value differs: The time-low and time-high parts (the first and third groups of hexadecimal digits, respectively) are swapped. This moves the more

rapidly varying part to the right and can improve indexing efficiency if the result is stored in an indexed column.

Time-part swapping assumes the use of UUID version 1 values, such as are generated by the `UUID()` function. For UUID values produced by other means that do not follow version 1 format, time-part swapping provides no benefit. For details about version 1 format, see the `UUID()` function description.

Suppose that you have the following string UUID value:

```
mysql> SET @uuid = '6ccd780c-baba-1026-9564-5b8c656024db';
```

To convert the string UUID to binary with or without time-part swapping, use `UUID_TO_BIN()`:

```
mysql> SELECT HEX(UUID_TO_BIN(@uuid));
+----------------------------------+
| HEX(UUID_TO_BIN(@uuid))          |
+----------------------------------+
| 6CCD780CBABA102695645B8C656024DB |
+----------------------------------+
mysql> SELECT HEX(UUID_TO_BIN(@uuid, 0));
+----------------------------------+
| HEX(UUID_TO_BIN(@uuid, 0))       |
+----------------------------------+
| 6CCD780CBABA102695645B8C656024DB |
+----------------------------------+
mysql> SELECT HEX(UUID_TO_BIN(@uuid, 1));
+----------------------------------+
| HEX(UUID_TO_BIN(@uuid, 1))       |
+----------------------------------+
| 1026BABA6CCD780C95645B8C656024DB |
+----------------------------------+
```

To convert a binary UUID returned by `UUID_TO_BIN()` to a string UUID, use `BIN_TO_UUID()`. If you produce a binary UUID by calling `UUID_TO_BIN()` with a second argument of 1 to swap time parts, you should also pass a second argument of 1 to `BIN_TO_UUID()` to unswap the time parts when converting the binary UUID back to a string UUID:

```
mysql> SELECT BIN_TO_UUID(UUID_TO_BIN(@uuid));
+--------------------------------------+
| BIN_TO_UUID(UUID_TO_BIN(@uuid))      |
+--------------------------------------+
| 6ccd780c-baba-1026-9564-5b8c656024db |
+--------------------------------------+
mysql> SELECT BIN_TO_UUID(UUID_TO_BIN(@uuid,0),0);
+--------------------------------------+
| BIN_TO_UUID(UUID_TO_BIN(@uuid,0),0)  |
+--------------------------------------+
| 6ccd780c-baba-1026-9564-5b8c656024db |
+--------------------------------------+
mysql> SELECT BIN_TO_UUID(UUID_TO_BIN(@uuid,1),1);
+--------------------------------------+
| BIN_TO_UUID(UUID_TO_BIN(@uuid,1),1)  |
+--------------------------------------+
| 6ccd780c-baba-1026-9564-5b8c656024db |
+--------------------------------------+
```

If the use of time-part swapping is not the same for the conversion in both directions, the original UUID is not recovered properly:

```
mysql> SELECT BIN_TO_UUID(UUID_TO_BIN(@uuid,0),1);
+--------------------------------------+
| BIN_TO_UUID(UUID_TO_BIN(@uuid,0),1)  |
+--------------------------------------+
| baba1026-780c-6ccd-9564-5b8c656024db |
```

```
+--------------------------------------+
mysql> SELECT BIN_TO_UUID(UUID_TO_BIN(@uuid,1),0);
+--------------------------------------+
| BIN_TO_UUID(UUID_TO_BIN(@uuid,1),0)  |
+--------------------------------------+
| 1026baba-6ccd-780c-9564-5b8c656024db |
+--------------------------------------+
```

If `UUID_TO_BIN()` is invoked from within the `mysql` client, binary strings display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

- `VALUES(col_name)`

  In an `INSERT ... ON DUPLICATE KEY UPDATE` statement, you can use the `VALUES(col_name)` function in the `UPDATE` clause to refer to column values from the `INSERT` portion of the statement. In other words, `VALUES(col_name)` in the `UPDATE` clause refers to the value of `col_name` that would be inserted, had no duplicate-key conflict occurred. This function is especially useful in multiple-row inserts. The `VALUES()` function is meaningful only in the `ON DUPLICATE KEY UPDATE` clause of `INSERT` statements and returns `NULL` otherwise. See Section 13.2.7.2, “INSERT ... ON DUPLICATE KEY UPDATE Statement”.

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3),(4,5,6)
    -> ON DUPLICATE KEY UPDATE c=VALUES(a)+VALUES(b);
```

**Important**

This usage is deprecated in MySQL 8.0.20, and is subject to removal in a future release of MySQL. Use a row alias, or row and column aliases, instead. See Section 13.2.7.2, “INSERT ... ON DUPLICATE KEY UPDATE Statement”, for more information and examples.

# 12.25 Precision Math

MySQL provides support for precision math: numeric value handling that results in extremely accurate results and a high degree control over invalid values. Precision math is based on these two features:

- SQL modes that control how strict the server is about accepting or rejecting invalid data.
- The MySQL library for fixed-point arithmetic.

These features have several implications for numeric operations and provide a high degree of compliance with standard SQL:

- **Precise calculations**: For exact-value numbers, calculations do not introduce floating-point errors. Instead, exact precision is used. For example, MySQL treats a number such as `.0001` as an exact value rather than as an approximation, and summing it 10,000 times produces a result of exactly `1`, not a value that is merely “close” to 1.
- **Well-defined rounding behavior**: For exact-value numbers, the result of `ROUND()` depends on its argument, not on environmental factors such as how the underlying C library works.
- **Platform independence**: Operations on exact numeric values are the same across different platforms such as Windows and Unix.
- **Control over handling of invalid values**: Overflow and division by zero are detectable and can be treated as errors. For example, you can treat a value that is too large for a column as an error rather than having the value truncated to lie within the range of the column's data type. Similarly, you can treat division by zero as an error rather than as an operation that produces a result of `NULL`. The choice of which approach to take is determined by the setting of the server SQL mode.

The following discussion covers several aspects of how precision math works, including possible incompatibilities with older applications. At the end, some examples are given that demonstrate how

MySQL handles numeric operations precisely. For information about controlling the SQL mode, see Section 5.1.11, “Server SQL Modes”.

## 12.25.1 Types of Numeric Values

The scope of precision math for exact-value operations includes the exact-value data types (integer and `DECIMAL` types) and exact-value numeric literals. Approximate-value data types and numeric literals are handled as floating-point numbers.

Exact-value numeric literals have an integer part or fractional part, or both. They may be signed. Examples: `1`, `.2`, `3.4`, `-5`, `-6.78`, `+9.10`.

Approximate-value numeric literals are represented in scientific notation with a mantissa and exponent. Either or both parts may be signed. Examples: `1.2E3`, `1.2E-3`, `-1.2E3`, `-1.2E-3`.

Two numbers that look similar may be treated differently. For example, `2.34` is an exact-value (fixed-point) number, whereas `2.34E0` is an approximate-value (floating-point) number.

The `DECIMAL` data type is a fixed-point type and calculations are exact. In MySQL, the `DECIMAL` type has several synonyms: `NUMERIC`, `DEC`, `FIXED`. The integer types also are exact-value types.

The `FLOAT` and `DOUBLE` data types are floating-point types and calculations are approximate. In MySQL, types that are synonymous with `FLOAT` or `DOUBLE` are `DOUBLE PRECISION` and `REAL`.

## 12.25.2 DECIMAL Data Type Characteristics

This section discusses the characteristics of the `DECIMAL` data type (and its synonyms), with particular regard to the following topics:

- Maximum number of digits
- Storage format
- Storage requirements
- The nonstandard MySQL extension to the upper range of `DECIMAL` columns

The declaration syntax for a `DECIMAL` column is `DECIMAL(M,D)`. The ranges of values for the arguments are as follows:

- *`M`* is the maximum number of digits (the precision). It has a range of 1 to 65.
- *`D`* is the number of digits to the right of the decimal point (the scale). It has a range of 0 to 30 and must be no larger than *`M`*.

If *`D`* is omitted, the default is 0. If *`M`* is omitted, the default is 10.

The maximum value of 65 for *`M`* means that calculations on `DECIMAL` values are accurate up to 65 digits. This limit of 65 digits of precision also applies to exact-value numeric literals, so the maximum range of such literals differs from before. (There is also a limit on how long the text of `DECIMAL` literals can be; see Section 12.25.3, “Expression Handling”.)

Values for `DECIMAL` columns are stored using a binary format that packs nine decimal digits into 4 bytes. The storage requirements for the integer and fractional parts of each value are determined separately. Each multiple of nine digits requires 4 bytes, and any remaining digits left over require some fraction of 4 bytes. The storage required for remaining digits is given by the following table.

| Leftover Digits | Number of Bytes |
|---|---|
| 0 | 0 |

| Leftover Digits | Number of Bytes |
|---|---|
| 1–2 | 1 |
| 3–4 | 2 |
| 5–6 | 3 |
| 7–9 | 4 |

For example, a `DECIMAL(18,9)` column has nine digits on either side of the decimal point, so the integer part and the fractional part each require 4 bytes. A `DECIMAL(20,6)` column has fourteen integer digits and six fractional digits. The integer digits require four bytes for nine of the digits and 3 bytes for the remaining five digits. The six fractional digits require 3 bytes.

`DECIMAL` columns do not store a leading `+` character or `-` character or leading `0` digits. If you insert `+0003.1` into a `DECIMAL(5,1)` column, it is stored as `3.1`. For negative numbers, a literal `-` character is not stored.

`DECIMAL` columns do not permit values larger than the range implied by the column definition. For example, a `DECIMAL(3,0)` column supports a range of `-999` to `999`. A `DECIMAL(M,D)` column permits up to `M` - `D` digits to the left of the decimal point.

The SQL standard requires that the precision of `NUMERIC(M,D)` be *exactly* `M` digits. For `DECIMAL(M,D)`, the standard requires a precision of at least `M` digits but permits more. In MySQL, `DECIMAL(M,D)` and `NUMERIC(M,D)` are the same, and both have a precision of exactly `M` digits.

For a full explanation of the internal format of `DECIMAL` values, see the file `strings/decimal.c` in a MySQL source distribution. The format is explained (with an example) in the `decimal2bin()` function.

## 12.25.3 Expression Handling

With precision math, exact-value numbers are used as given whenever possible. For example, numbers in comparisons are used exactly as given without a change in value. In strict SQL mode, for `INSERT` into a column with an exact data type (`DECIMAL` or integer), a number is inserted with its exact value if it is within the column range. When retrieved, the value should be the same as what was inserted. (If strict SQL mode is not enabled, truncation for `INSERT` is permissible.)

Handling of a numeric expression depends on what kind of values the expression contains:

- If any approximate values are present, the expression is approximate and is evaluated using floating-point arithmetic.
- If no approximate values are present, the expression contains only exact values. If any exact value contains a fractional part (a value following the decimal point), the expression is evaluated using `DECIMAL` exact arithmetic and has a precision of 65 digits. The term “exact” is subject to the limits of what can be represented in binary. For example, `1.0/3.0` can be approximated in decimal notation as `.333...`, but not written as an exact number, so `(1.0/3.0)*3.0` does not evaluate to exactly `1.0`.
- Otherwise, the expression contains only integer values. The expression is exact and is evaluated using integer arithmetic and has a precision the same as `BIGINT` (64 bits).

If a numeric expression contains any strings, they are converted to double-precision floating-point values and the expression is approximate.

Inserts into numeric columns are affected by the SQL mode, which is controlled by the `sql_mode` system variable. (See Section 5.1.11, “Server SQL Modes”.) The following discussion mentions strict mode (selected by the `STRICT_ALL_TABLES` or `STRICT_TRANS_TABLES` mode values) and `ERROR_FOR_DIVISION_BY_ZERO`. To turn on all restrictions, you can simply use `TRADITIONAL` mode, which includes both strict mode values and `ERROR_FOR_DIVISION_BY_ZERO`:

```
SET sql_mode='TRADITIONAL';
```

If a number is inserted into an exact type column (`DECIMAL` or integer), it is inserted with its exact value if it is within the column range and precision.

If the value has too many digits in the fractional part, rounding occurs and a note is generated. Rounding is done as described in Section 12.25.4, “Rounding Behavior”. Truncation due to rounding of the fractional part is not an error, even in strict mode.

If the value has too many digits in the integer part, it is too large (out of range) and is handled as follows:

- If strict mode is not enabled, the value is truncated to the nearest legal value and a warning is generated.
- If strict mode is enabled, an overflow error occurs.

Prior to MySQL 8.0.31, for `DECIMAL` literals, in addition to the precision limit of 65 digits, there is a limit on how long the text of the literal can be. If the value exceeds approximately 80 characters, unexpected results can occur. For example:

```
mysql> SELECT
       CAST(00000000000000000000000000000000000000000000000000000000000000000000000000000020.01 AS DECIMA
+------------------+
| val              |
+------------------+
| 9999999999999.99 |
+------------------+
1 row in set, 2 warnings (0.00 sec)

mysql> SHOW WARNINGS;
+---------+------+---------------------------------------------+
| Level   | Code | Message                                     |
+---------+------+---------------------------------------------+
| Warning | 1292 | Truncated incorrect DECIMAL value: '20'     |
| Warning | 1264 | Out of range value for column 'val' at row 1 |
+---------+------+---------------------------------------------+
2 rows in set (0.00 sec)
```

As of MySQL 8.0.31, this should no longer be an issue, as shown here:

```
mysql> SELECT
       CAST(00000000000000000000000000000000000000000000000000000000000000000000000000000020.01 AS DECIMA
+-------+
| val   |
+-------+
| 20.01 |
+-------+
1 row in set (0.00 sec)
```

Underflow is not detected, so underflow handling is undefined.

For inserts of strings into numeric columns, conversion from string to number is handled as follows if the string has nonnumeric contents:

- A string that does not begin with a number cannot be used as a number and produces an error in strict mode, or a warning otherwise. This includes the empty string.
- A string that begins with a number can be converted, but the trailing nonnumeric portion is truncated. If the truncated portion contains anything other than spaces, this produces an error in strict mode, or a warning otherwise.

By default, division by zero produces a result of `NULL` and no warning. By setting the SQL mode appropriately, division by zero can be restricted.

With the `ERROR_FOR_DIVISION_BY_ZERO` SQL mode enabled, MySQL handles division by zero differently:

- If strict mode is not enabled, a warning occurs.
- If strict mode is enabled, inserts and updates involving division by zero are prohibited, and an error occurs.

In other words, inserts and updates involving expressions that perform division by zero can be treated as errors, but this requires `ERROR_FOR_DIVISION_BY_ZERO` in addition to strict mode.

Suppose that we have this statement:

```
INSERT INTO t SET i = 1/0;
```

This is what happens for combinations of strict and `ERROR_FOR_DIVISION_BY_ZERO` modes.

| **`sql_mode` Value** | **Result** |
|---|---|
| `''` (Default) | No warning, no error; `i` is set to `NULL`. |
| strict | No warning, no error; `i` is set to `NULL`. |
| `ERROR_FOR_DIVISION_BY_ZERO` | Warning, no error; `i` is set to `NULL`. |
| strict,`ERROR_FOR_DIVISION_BY_ZERO` | Error condition; no row is inserted. |

## 12.25.4 Rounding Behavior

This section discusses precision math rounding for the `ROUND()` function and for inserts into columns with exact-value types (`DECIMAL` and integer).

The `ROUND()` function rounds differently depending on whether its argument is exact or approximate:

- For exact-value numbers, `ROUND()` uses the “round half up” rule: A value with a fractional part of .5 or greater is rounded up to the next integer if positive or down to the next integer if negative. (In other words, it is rounded away from zero.) A value with a fractional part less than .5 is rounded down to the next integer if positive or up to the next integer if negative. (In other words, it is rounded toward zero.)
- For approximate-value numbers, the result depends on the C library. On many systems, this means that `ROUND()` uses the “round to nearest even” rule: A value with a fractional part exactly half way between two integers is rounded to the nearest even integer.

The following example shows how rounding differs for exact and approximate values:

```
mysql> SELECT ROUND(2.5), ROUND(25E-1);
+------------+--------------+
| ROUND(2.5) | ROUND(25E-1) |
+------------+--------------+
| 3          |            2 |
+------------+--------------+
```

For inserts into a `DECIMAL` or integer column, the target is an exact data type, so rounding uses “round half away from zero,” regardless of whether the value to be inserted is exact or approximate:

```
mysql> CREATE TABLE t (d DECIMAL(10,0));
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t VALUES(2.5),(2.5E0);
Query OK, 2 rows affected, 2 warnings (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 2

mysql> SHOW WARNINGS;
+-------+------+----------------------------------------+
| Level | Code | Message                                |
+-------+------+----------------------------------------+
| Note  | 1265 | Data truncated for column 'd' at row 1 |
| Note  | 1265 | Data truncated for column 'd' at row 2 |
```

```
+-------+------+----------------------------------------+
2 rows in set (0.00 sec)

mysql> SELECT d FROM t;
+------+
| d    |
+------+
|    3 |
|    3 |
+------+
2 rows in set (0.00 sec)
```

The `SHOW WARNINGS` statement displays the notes that are generated by truncation due to rounding of the fractional part. Such truncation is not an error, even in strict SQL mode (see Section 12.25.3, “Expression Handling”).

## 12.25.5 Precision Math Examples

This section provides some examples that show precision math query results in MySQL. These examples demonstrate the principles described in Section 12.25.3, “Expression Handling”, and Section 12.25.4, “Rounding Behavior”.

**Example 1**. Numbers are used with their exact value as given when possible:

```
mysql> SELECT (.1 + .2) = .3;
+----------------+
| (.1 + .2) = .3 |
+----------------+
|              1 |
+----------------+
```

For floating-point values, results are inexact:

```
mysql> SELECT (.1E0 + .2E0) = .3E0;
+----------------------+
| (.1E0 + .2E0) = .3E0 |
+----------------------+
|                    0 |
+----------------------+
```

Another way to see the difference in exact and approximate value handling is to add a small number to a sum many times. Consider the following stored procedure, which adds `.0001` to a variable 1,000 times.

```
CREATE PROCEDURE p ()
BEGIN
  DECLARE i INT DEFAULT 0;
  DECLARE d DECIMAL(10,4) DEFAULT 0;
  DECLARE f FLOAT DEFAULT 0;
  WHILE i < 10000 DO
    SET d = d + .0001;
    SET f = f + .0001E0;
    SET i = i + 1;
  END WHILE;
  SELECT d, f;
END;
```

The sum for both `d` and `f` logically should be 1, but that is true only for the decimal calculation. The floating-point calculation introduces small errors:

```
+--------+------------------+
| d      | f                |
+--------+------------------+
| 1.0000 | 0.99999999999991 |
+--------+------------------+
```

**Example 2**. Multiplication is performed with the scale required by standard SQL. That is, for two numbers *`X1`* and *`X2`* that have scale *`S1`* and *`S2`*, the scale of the result is *`S1 + S2`*:

```
mysql> SELECT .01 * .01;
+-----------+
| .01 * .01 |
+-----------+
| 0.0001    |
+-----------+
```

**Example 3**. Rounding behavior for exact-value numbers is well-defined:

Rounding behavior (for example, with the `ROUND()` function) is independent of the implementation of the underlying C library, which means that results are consistent from platform to platform.

- Rounding for exact-value columns (`DECIMAL` and integer) and exact-valued numbers uses the "round half away from zero" rule. A value with a fractional part of .5 or greater is rounded away from zero to the nearest integer, as shown here:

  ```
  mysql> SELECT ROUND(2.5), ROUND(-2.5);
  +------------+-------------+
  | ROUND(2.5) | ROUND(-2.5) |
  +------------+-------------+
  | 3          | -3          |
  +------------+-------------+
  ```

- Rounding for floating-point values uses the C library, which on many systems uses the "round to nearest even" rule. A value with a fractional part exactly half way between two integers is rounded to the nearest even integer:

  ```
  mysql> SELECT ROUND(2.5E0), ROUND(-2.5E0);
  +--------------+---------------+
  | ROUND(2.5E0) | ROUND(-2.5E0) |
  +--------------+---------------+
  |            2 |            -2 |
  +--------------+---------------+
  ```

**Example 4**. In strict mode, inserting a value that is out of range for a column causes an error, rather than truncation to a legal value.

When MySQL is not running in strict mode, truncation to a legal value occurs:

```
mysql> SET sql_mode='';
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE t (i TINYINT);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO t SET i = 128;
Query OK, 1 row affected, 1 warning (0.00 sec)

mysql> SELECT i FROM t;
+------+
| i    |
+------+
|  127 |
+------+
1 row in set (0.00 sec)
```

However, an error occurs if strict mode is in effect:

```
mysql> SET sql_mode='STRICT_ALL_TABLES';
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE t (i TINYINT);
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t SET i = 128;
ERROR 1264 (22003): Out of range value adjusted for column 'i' at row 1

mysql> SELECT i FROM t;
Empty set (0.00 sec)
```

**Example 5**: In strict mode and with `ERROR_FOR_DIVISION_BY_ZERO` set, division by zero causes an error, not a result of `NULL`.

In nonstrict mode, division by zero has a result of `NULL`:

```
mysql> SET sql_mode='';
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE t (i TINYINT);
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t SET i = 1 / 0;
Query OK, 1 row affected (0.00 sec)

mysql> SELECT i FROM t;
+------+
| i    |
+------+
| NULL |
+------+
1 row in set (0.03 sec)
```

However, division by zero is an error if the proper SQL modes are in effect:

```
mysql> SET sql_mode='STRICT_ALL_TABLES,ERROR_FOR_DIVISION_BY_ZERO';
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE t (i TINYINT);
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t SET i = 1 / 0;
ERROR 1365 (22012): Division by 0

mysql> SELECT i FROM t;
Empty set (0.01 sec)
```

**Example 6**. Exact-value literals are evaluated as exact values.

Approximate-value literals are evaluated using floating point, but exact-value literals are handled as `DECIMAL`:

```
mysql> CREATE TABLE t SELECT 2.5 AS a, 25E-1 AS b;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> DESCRIBE t;
+-------+----------------------+------+-----+---------+-------+
| Field | Type                 | Null | Key | Default | Extra |
+-------+----------------------+------+-----+---------+-------+
| a     | decimal(2,1) unsigned | NO   |     | 0.0     |       |
| b     | double               | NO   |     | 0       |       |
+-------+----------------------+------+-----+---------+-------+
2 rows in set (0.01 sec)
```

**Example 7**. If the argument to an aggregate function is an exact numeric type, the result is also an exact numeric type, with a scale at least that of the argument.

Consider these statements:

```
mysql> CREATE TABLE t (i INT, d DECIMAL, f FLOAT);
mysql> INSERT INTO t VALUES(1,1,1);
mysql> CREATE TABLE y SELECT AVG(i), AVG(d), AVG(f) FROM t;
```

The result is a double only for the floating-point argument. For exact type arguments, the result is also an exact type:

```
mysql> DESCRIBE y;
+--------+---------------+------+-----+---------+-------+
| Field  | Type          | Null | Key | Default | Extra |
```

```
+--------+---------------+------+-----+---------+-------+
| AVG(i) | decimal(14,4) | YES  |     | NULL    |       |
| AVG(d) | decimal(14,4) | YES  |     | NULL    |       |
| AVG(f) | double        | YES  |     | NULL    |       |
+--------+---------------+------+-----+---------+-------+
```

The result is a double only for the floating-point argument. For exact type arguments, the result is also an exact type.

# Chapter 13 SQL Statements

## Table of Contents

This chapter describes the syntax for the SQL statements supported by MySQL.

# 13.1 Data Definition Statements

## 13.1.1 Atomic Data Definition Statement Support

MySQL 8.0 supports atomic Data Definition Language (DDL) statements. This feature is referred to as *atomic DDL*. An atomic DDL statement combines the data dictionary updates, storage engine operations, and binary log writes associated with a DDL operation into a single, atomic operation. The

operation is either committed, with applicable changes persisted to the data dictionary, storage engine, and binary log, or is rolled back, even if the server halts during the operation.

**Note**

*Atomic DDL* is not *transactional DDL*. DDL statements, atomic or otherwise, implicitly end any transaction that is active in the current session, as if you had done a `COMMIT` before executing the statement. This means that DDL statements cannot be performed within another transaction, within transaction control statements such as `START TRANSACTION ... COMMIT`, or combined with other statements within the same transaction.

Atomic DDL is made possible by the introduction of the MySQL data dictionary in MySQL 8.0. In earlier MySQL versions, metadata was stored in metadata files, nontransactional tables, and storage engine-specific dictionaries, which necessitated intermediate commits. Centralized, transactional metadata storage provided by the MySQL data dictionary removed this barrier, making it possible to restructure DDL statement operations to be atomic.

The atomic DDL feature is described under the following topics in this section:

- Supported DDL Statements
- Atomic DDL Characteristics
- Changes in DDL Statement Behavior
- Storage Engine Support
- Viewing DDL Logs

## Supported DDL Statements

The atomic DDL feature supports both table and non-table DDL statements. Table-related DDL operations require storage engine support, whereas non-table DDL operations do not. Currently, only the `InnoDB` storage engine supports atomic DDL.

- Supported table DDL statements include `CREATE`, `ALTER`, and `DROP` statements for databases, tablespaces, tables, and indexes, and the `TRUNCATE TABLE` statement.
- Supported non-table DDL statements include:
  - `CREATE` and `DROP` statements, and, if applicable, `ALTER` statements for stored programs, triggers, views, and loadable functions.
  - Account management statements: `CREATE`, `ALTER`, `DROP`, and, if applicable, `RENAME` statements for users and roles, as well as `GRANT` and `REVOKE` statements.

The following statements are not supported by the atomic DDL feature:

- Table-related DDL statements that involve a storage engine other than `InnoDB`.
- `INSTALL PLUGIN` and `UNINSTALL PLUGIN` statements.
- `INSTALL COMPONENT` and `UNINSTALL COMPONENT` statements.
- `CREATE SERVER`, `ALTER SERVER`, and `DROP SERVER` statements.

## Atomic DDL Characteristics

The characteristics of atomic DDL statements include the following:

- Metadata updates, binary log writes, and storage engine operations, where applicable, are combined into a single atomic operation.
- There are no intermediate commits at the SQL layer during the DDL operation.

- Where applicable:
  - The state of data dictionary, routine, event, and loadable function caches is consistent with the status of the DDL operation, meaning that caches are updated to reflect whether or not the DDL operation was completed successfully or rolled back.
  - The storage engine methods involved in a DDL operation do not perform intermediate commits, and the storage engine registers itself as part of the DDL operation.
  - The storage engine supports redo and rollback of DDL operations, which is performed in the *Post-DDL* phase of the DDL operation.
- The visible behaviour of DDL operations is atomic, which changes the behavior of some DDL statements. See Changes in DDL Statement Behavior.

## Changes in DDL Statement Behavior

This section describes changes in DDL statement behavior due to the introduction of atomic DDL support.

- `DROP TABLE` operations are fully atomic if all named tables use an atomic DDL-supported storage engine. The statement either drops all tables successfully or is rolled back.

  `DROP TABLE` fails with an error if a named table does not exist, and no changes are made, regardless of the storage engine. This change in behavior is demonstrated in the following example, where the `DROP TABLE` statement fails because a named table does not exist:

```
mysql> CREATE TABLE t1 (c1 INT);
mysql> DROP TABLE t1, t2;
ERROR 1051 (42S02): Unknown table 'test.t2'
mysql> SHOW TABLES;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
```

  Prior to the introduction of atomic DDL, `DROP TABLE` reports an error for the named table that does not exist but succeeds for the named table that does exist:

```
mysql> CREATE TABLE t1 (c1 INT);
mysql> DROP TABLE t1, t2;
ERROR 1051 (42S02): Unknown table 'test.t2'
mysql> SHOW TABLES;
Empty set (0.00 sec)
```

  

  **Note**

  Due to this change in behavior, a partially completed `DROP TABLE` statement on a MySQL 5.7 replication source server fails when replicated on a MySQL 8.0 replica. To avoid this failure scenario, use `IF EXISTS` syntax in `DROP TABLE` statements to prevent errors from occurring for tables that do not exist.

- `DROP DATABASE` is atomic if all tables use an atomic DDL-supported storage engine. The statement either drops all objects successfully or is rolled back. However, removal of the database directory from the file system occurs last and is not part of the atomic operation. If removal of the database directory fails due to a file system error or server halt, the `DROP DATABASE` transaction is not rolled back.
- For tables that do not use an atomic DDL-supported storage engine, table deletion occurs outside of the atomic `DROP TABLE` or `DROP DATABASE` transaction. Such table deletions are written to the binary log individually, which limits the discrepancy between the storage engine, data dictionary, and binary log to one table at most in the case of an interrupted `DROP TABLE` or `DROP DATABASE`

operation. For operations that drop multiple tables, the tables that do not use an atomic DDL-supported storage engine are dropped before tables that do.

- `CREATE TABLE`, `ALTER TABLE`, `RENAME TABLE`, `TRUNCATE TABLE`, `CREATE TABLESPACE`, and `DROP TABLESPACE` operations for tables that use an atomic DDL-supported storage engine are either fully committed or rolled back if the server halts during their operation. In earlier MySQL releases, interruption of these operations could cause discrepancies between the storage engine, data dictionary, and binary log, or leave behind orphan files. `RENAME TABLE` operations are only atomic if all named tables use an atomic DDL-supported storage engine.

- As of MySQL 8.0.21, on storage engines that support atomic DDL, the `CREATE TABLE ... SELECT` statement is logged as one transaction in the binary log when row-based replication is in use. Previously, it was logged as two transactions, one to create the table, and the other to insert data. A server failure between the two transactions or while inserting data could result in replication of an empty table. With the introduction of atomic DDL support, `CREATE TABLE ... SELECT` statements are now safe for row-based replication and permitted for use with GTID-based replication.

  On storage engines that support both atomic DDL and foreign key constraints, creation of foreign keys is not permitted in `CREATE TABLE ... SELECT` statements when row-based replication is in use. Foreign key constraints can be added later using `ALTER TABLE`.

  When `CREATE TABLE ... SELECT` is applied as an atomic operation, a metadata lock is held on the table while data is inserted, which prevents concurrent access to the table for the duration of the operation.

- `DROP VIEW` fails if a named view does not exist, and no changes are made. The change in behavior is demonstrated in this example, where the `DROP VIEW` statement fails because a named view does not exist:

```
mysql> CREATE VIEW test.viewA AS SELECT * FROM t;
mysql> DROP VIEW test.viewA, test.viewB;
ERROR 1051 (42S02): Unknown table 'test.viewB'
mysql> SHOW FULL TABLES IN test WHERE TABLE_TYPE LIKE 'VIEW';
+----------------+------------+
| Tables_in_test | Table_type |
+----------------+------------+
| viewA          | VIEW       |
+----------------+------------+
```

  Prior to the introduction of atomic DDL, `DROP VIEW` returns an error for the named view that does not exist but succeeds for the named view that does exist:

```
mysql> CREATE VIEW test.viewA AS SELECT * FROM t;
mysql> DROP VIEW test.viewA, test.viewB;
ERROR 1051 (42S02): Unknown table 'test.viewB'
mysql> SHOW FULL TABLES IN test WHERE TABLE_TYPE LIKE 'VIEW';
Empty set (0.00 sec)
```

**Note**

Due to this change in behavior, a partially completed `DROP VIEW` operation on a MySQL 5.7 replication source server fails when replicated on a MySQL 8.0 replica. To avoid this failure scenario, use `IF EXISTS` syntax in `DROP VIEW` statements to prevent an error from occurring for views that do not exist.

- Partial execution of account management statements is no longer permitted. Account management statements either succeed for all named users or roll back and have no effect if an error occurs. In

earlier MySQL versions, account management statements that name multiple users could succeed for some users and fail for others.

The change in behavior is demonstrated in this example, where the second `CREATE USER` statement returns an error but fails because it cannot succeed for all named users.

```
mysql> CREATE USER userA;
mysql> CREATE USER userA, userB;
ERROR 1396 (HY000): Operation CREATE USER failed for 'userA'@'%'
mysql> SELECT User FROM mysql.user WHERE User LIKE 'user%';
+-------+
| User  |
+-------+
| userA |
+-------+
```

Prior to the introduction of atomic DDL, the second `CREATE USER` statement returns an error for the named user that does not exist but succeeds for the named user that does exist:

```
mysql> CREATE USER userA;
mysql> CREATE USER userA, userB;
ERROR 1396 (HY000): Operation CREATE USER failed for 'userA'@'%'
mysql> SELECT User FROM mysql.user WHERE User LIKE 'user%';
+-------+
| User  |
+-------+
| userA |
| userB |
+-------+
```

**Note**

Due to this change in behavior, partially completed account management statements on a MySQL 5.7 replication source server fail when replicated on a MySQL 8.0 replica. To avoid this failure scenario, use `IF EXISTS` or `IF NOT EXISTS` syntax, as appropriate, in account management statements to prevent errors related to named users.

## Storage Engine Support

Currently, only the `InnoDB` storage engine supports atomic DDL. Storage engines that do not support atomic DDL are exempted from DDL atomicity. DDL operations involving exempted storage engines remain capable of introducing inconsistencies that can occur when operations are interrupted or only partially completed.

To support redo and rollback of DDL operations, `InnoDB` writes DDL logs to the `mysql.innodb_ddl_log` table, which is a hidden data dictionary table that resides in the `mysql.ibd` data dictionary tablespace.

To view DDL logs that are written to the `mysql.innodb_ddl_log` table during a DDL operation, enable the `innodb_print_ddl_logs` configuration option. For more information, see Viewing DDL Logs.

**Note**

The redo logs for changes to the `mysql.innodb_ddl_log` table are flushed to disk immediately regardless of the `innodb_flush_log_at_trx_commit` setting. Flushing the redo logs immediately avoids situations where data files are modified by DDL operations but the redo logs for changes to the `mysql.innodb_ddl_log` table resulting from those operations are not persisted to disk. Such a situation could cause errors during rollback or recovery.

The InnoDB storage engine executes DDL operations in phases. DDL operations such as ALTER TABLE may perform the *Prepare* and *Perform* phases multiple times prior to the *Commit* phase.

1. *Prepare*: Create the required objects and write the DDL logs to the mysql.innodb_ddl_log table. The DDL logs define how to roll forward and roll back the DDL operation.
2. *Perform*: Perform the DDL operation. For example, perform a create routine for a CREATE TABLE operation.
3. *Commit*: Update the data dictionary and commit the data dictionary transaction.
4. *Post-DDL*: Replay and remove DDL logs from the mysql.innodb_ddl_log table. To ensure that rollback can be performed safely without introducing inconsistencies, file operations such as renaming or removing data files are performed in this final phase. This phase also removes dynamic metadata from the mysql.innodb_dynamic_metadata data dictionary table for DROP TABLE, TRUNCATE TABLE, and other DDL operations that rebuild the table.

DDL logs are replayed and removed from the mysql.innodb_ddl_log table during the *Post-DDL* phase, regardless of whether the DDL operation is committed or rolled back. DDL logs should only remain in the mysql.innodb_ddl_log table if the server is halted during a DDL operation. In this case, the DDL logs are replayed and removed after recovery.

In a recovery situation, a DDL operation may be committed or rolled back when the server is restarted. If the data dictionary transaction that was performed during the *Commit* phase of a DDL operation is present in the redo log and binary log, the operation is considered successful and is rolled forward. Otherwise, the incomplete data dictionary transaction is rolled back when InnoDB replays data dictionary redo logs, and the DDL operation is rolled back.

## Viewing DDL Logs

To view DDL logs that are written to the mysql.innodb_ddl_log data dictionary table during atomic DDL operations that involve the InnoDB storage engine, enable innodb_print_ddl_logs to have MySQL write the DDL logs to stderr. Depending on the host operating system and MySQL configuration, stderr may be the error log, terminal, or console window. See Section 5.4.2.2, “Default Error Log Destination Configuration”.

InnoDB writes DDL logs to the mysql.innodb_ddl_log table to support redo and rollback of DDL operations. The mysql.innodb_ddl_log table is a hidden data dictionary table that resides in the mysql.ibd data dictionary tablespace. Like other hidden data dictionary tables, the mysql.innodb_ddl_log table cannot be accessed directly in non-debug versions of MySQL. (See Section 14.1, “Data Dictionary Schema”.) The structure of the mysql.innodb_ddl_log table corresponds to this definition:

```
CREATE TABLE mysql.innodb_ddl_log (
  id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  thread_id BIGINT UNSIGNED NOT NULL,
  type INT UNSIGNED NOT NULL,
  space_id INT UNSIGNED,
  page_no INT UNSIGNED,
  index_id BIGINT UNSIGNED,
  table_id BIGINT UNSIGNED,
  old_file_path VARCHAR(512) COLLATE utf8mb4_bin,
  new_file_path VARCHAR(512) COLLATE utf8mb4_bin,
  KEY(thread_id)
);
```

- id: A unique identifier for a DDL log record.
- thread_id: Each DDL log record is assigned a thread_id, which is used to replay and remove DDL logs that belong to a particular DDL operation. DDL operations that involve multiple data file operations generate multiple DDL log records.

- `type`: The DDL operation type. Types include `FREE` (drop an index tree), `DELETE` (delete a file), `RENAME` (rename a file), or `DROP` (drop metadata from the `mysql.innodb_dynamic_metadata` data dictionary table).
- `space_id`: The tablespace ID.
- `page_no`: A page that contains allocation information; an index tree root page, for example.
- `index_id`: The index ID.
- `table_id`: The table ID.
- `old_file_path`: The old tablespace file path. Used by DDL operations that create or drop tablespace files; also used by DDL operations that rename a tablespace.
- `new_file_path`: The new tablespace file path. Used by DDL operations that rename tablespace files.

This example demonstrates enabling `innodb_print_ddl_logs` to view DDL logs written to `strderr` for a `CREATE TABLE` operation.

```
mysql> SET GLOBAL innodb_print_ddl_logs=1;
mysql> CREATE TABLE t1 (c1 INT) ENGINE = InnoDB;
```

```
[Note] [000000] InnoDB: DDL log insert : [DDL record: DELETE SPACE, id=18, thread_id=7,
space_id=5, old_file_path=./test/t1.ibd]
[Note] [000000] InnoDB: DDL log delete : by id 18
[Note] [000000] InnoDB: DDL log insert : [DDL record: REMOVE CACHE, id=19, thread_id=7,
table_id=1058, new_file_path=test/t1]
[Note] [000000] InnoDB: DDL log delete : by id 19
[Note] [000000] InnoDB: DDL log insert : [DDL record: FREE, id=20, thread_id=7,
space_id=5, index_id=132, page_no=4]
[Note] [000000] InnoDB: DDL log delete : by id 20
[Note] [000000] InnoDB: DDL log post ddl : begin for thread id : 7
[Note] [000000] InnoDB: DDL log post ddl : end for thread id : 7
```

## 13.1.2 ALTER DATABASE Statement

```
ALTER {DATABASE | SCHEMA} [db_name]
    alter_option ...

alter_option: {
    [DEFAULT] CHARACTER SET [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
  | [DEFAULT] ENCRYPTION [=] {'Y' | 'N'}
  | READ ONLY [=] {DEFAULT | 0 | 1}
}
```

`ALTER DATABASE` enables you to change the overall characteristics of a database. These characteristics are stored in the data dictionary. This statement requires the `ALTER` privilege on the database. `ALTER SCHEMA` is a synonym for `ALTER DATABASE`.

If the database name is omitted, the statement applies to the default database. In that case, an error occurs if there is no default database.

For any *`alter_option`* omitted from the statement, the database retains its current option value, with the exception that changing the character set may change the collation and vice versa.

- Character Set and Collation Options
- Encryption Option
- Read Only Option

## Character Set and Collation Options

The `CHARACTER SET` option changes the default database character set. The `COLLATE` option changes the default database collation. For information about character set and collation names, see Chapter 10, *Character Sets, Collations, Unicode*.

To see the available character sets and collations, use the `SHOW CHARACTER SET` and `SHOW COLLATION` statements, respectively. See Section 13.7.7.3, “SHOW CHARACTER SET Statement”, and Section 13.7.7.4, “SHOW COLLATION Statement”.

A stored routine that uses the database defaults when the routine is created includes those defaults as part of its definition. (In a stored routine, variables with character data types use the database defaults if the character set or collation are not specified explicitly. See Section 13.1.17, “CREATE PROCEDURE and CREATE FUNCTION Statements”.) If you change the default character set or collation for a database, any stored routines that are to use the new defaults must be dropped and recreated.

## Encryption Option

The `ENCRYPTION` option, introduced in MySQL 8.0.16, defines the default database encryption, which is inherited by tables created in the database. The permitted values are `'Y'` (encryption enabled) and `'N'` (encryption disabled).

The `mysql` system schema cannot be set to default encryption. The existing tables within it are part of the general `mysql` tablespace, which may be encrypted. The `information_schema` contains only views. It is not possible to create any tables within it. There is nothing on the disk to encrypt. All tables in the `performance_schema` use the `PERFORMANCE_SCHEMA` engine, which is purely in-memory. It is not possible to create any other tables in it. There is nothing on the disk to encrypt.

Only newly created tables inherit the default database encryption. For existing tables associated with the database, their encryption remains unchanged. If the `table_encryption_privilege_check` system variable is enabled, the `TABLE_ENCRYPTION_ADMIN` privilege is required to specify a default encryption setting that differs from the value of the `default_table_encryption` system variable. For more information, see Defining an Encryption Default for Schemas and General Tablespaces.

## Read Only Option

The `READ ONLY` option, introduced in MySQL 8.0.22, controls whether to permit modification of the database and objects within it. The permitted values are `DEFAULT` or `0` (not read only) and `1` (read only). This option is useful for database migration because a database for which `READ ONLY` is enabled can be migrated to another MySQL instance without concern that the database might be changed during the operation.

With NDB Cluster, making a database read only on one `mysqld` server is synchronized to other `mysqld` servers in the same cluster, so that the database becomes read only on all `mysqld` servers.

The `READ ONLY` option, if enabled, is displayed in the `INFORMATION_SCHEMA SCHEMATA_EXTENSIONS` table. See Section 26.3.32, “The INFORMATION_SCHEMA SCHEMATA_EXTENSIONS Table”.

The `READ ONLY` option cannot be enabled for these system schemas: `mysql`, `information_schema`, `performance_schema`.

In `ALTER DATABASE` statements, the `READ ONLY` option interacts with other instances of itself and with other options as follows:

- An error occurs if multiple instances of `READ ONLY` conflict (for example, `READ ONLY = 1 READ ONLY = 0`).
- An `ALTER DATABASE` statement that contains only (nonconflicting) `READ ONLY` options is permitted even for a read-only database.

- A mix of (nonconflicting) `READ ONLY` options with other options is permitted if the read-only state of the database either before or after the statement permits modifications. If the read-only state both before and after prohibits changes, an error occurs.

  This statement succeeds whether or not the database is read only:

  ```
  ALTER DATABASE mydb READ ONLY = 0 DEFAULT COLLATE utf8mb4_bin;
  ```

  This statement succeeds if the database is not read only, but fails if it is already read only:

  ```
  ALTER DATABASE mydb READ ONLY = 1 DEFAULT COLLATE utf8mb4_bin;
  ```

Enabling `READ ONLY` affects all users of the database, with these exceptions that are not subject to read-only checks:

- Statements executed by the server as part of server initialization, restart, upgrade, or replication.
- Statements in a file named at server startup by the `init_file` system variable.
- `TEMPORARY` tables; it is possible to create, alter, drop, and write to `TEMPORARY` tables in a read-only database.
- NDB Cluster non-SQL inserts and updates.

Other than for the excepted operations just listed, enabling `READ ONLY` prohibits write operations to the database and its objects, including their definitions, data, and metadata. The following list details affected SQL statements and operations:

- The database itself:
  - `CREATE DATABASE`
  - `ALTER DATABASE` (except to change the `READ ONLY` option)
  - `DROP DATABASE`
- Views:
  - `CREATE VIEW`
  - `ALTER VIEW`
  - `DROP VIEW`
  - Selecting from views that invoke functions with side effects.
  - Updating updatable views.
  - Statements that create or drop objects in a writable database are rejected if they affect metadata of a view in a read-only database (for example, by making the view valid or invalid).
- Stored routines:
  - `CREATE PROCEDURE`
  - `DROP PROCEDURE`
  - `CALL` (of procedures with side effects)
  - `CREATE FUNCTION`
  - `DROP FUNCTION`
  - `SELECT` (of functions with side effects)

    - For procedures and functions, read-only checks follow prelocking behavior. For `CALL` statements, read-only checks are done on a per-statement basis, so if some conditionally executed statement writing to a read-only database does not actually execute, the call still succeeds. On the other hand, for a function called within a `SELECT`, execution of the function body happens in prelocked mode. As long as a some statement within the function writes to a read-only database, execution of the function fails with an error regardless of whether the statement actually executes.

- Triggers:
  - `CREATE TRIGGER`
  - `DROP TRIGGER`
  - Trigger invocation.
- Events:
  - `CREATE EVENT`
  - `ALTER EVENT`
  - `DROP EVENT`
  - Event execution:
    - Executing an event in the database fails because that would change the last-execution timestamp, which is event metadata stored in the data dictionary. Failure of event execution also has the effect of causing the event scheduler to stop.
    - If an event writes to an object in a read-only database, execution of the event fails with an error, but the event scheduler is not stopped.

- Tables:
  - `CREATE TABLE`
  - `ALTER TABLE`
  - `CREATE INDEX`
  - `DROP INDEX`
  - `RENAME TABLE`
  - `TRUNCATE TABLE`
  - `DROP TABLE`
  - `DELETE`
  - `INSERT`
  - `IMPORT TABLE`
  - `LOAD DATA`
  - `LOAD XML`
  - `REPLACE`
  - `UPDATE`
  - For cascading foreign keys where the child table is in a read-only database, updates and deletes on the parent are rejected even if the child table is not directly affected.
  - For a `MERGE` table such as `CREATE TABLE s1.t(i int) ENGINE MERGE UNION (s2.t, s3.t), INSERT_METHOD=...`, the following behavior applies:
    - Inserting into the `MERGE` table (`INSERT into s1.t`) fails if at least one of `s1`, `s2`, `s3` is read only, regardless of insert method. The insert is refused even if it would actually end up in a writable table.
    - Dropping the `MERGE` table (`DROP TABLE s1.t`) succeeds as long as `s1` is not read only. It is permitted to drop a `MERGE` table that refers to a read-only database.

An `ALTER DATABASE` statement blocks until all concurrent transactions that have already accessed an object in the database being altered have committed. Conversely, a write transaction accessing an object in a database being altered in a concurrent `ALTER DATABASE` blocks until the `ALTER DATABASE` has committed.

If the Clone plugin is used to clone a local or remote data directory, the databases in the clone retain the read-only state they had in the source data directory. The read-only state does not affect the cloning process itself. If it is not desirable to have the same database read-only state in the clone, the option must be changed explicitly for the clone after the cloning process has finished, using `ALTER DATABASE` operations on the clone.

When cloning from a donor to a recipient, if the recipient has a user database that is read only, cloning fails with an error message. Cloning may be retried after making the database writable.

`READ ONLY` is permitted for `ALTER DATABASE`, but not for `CREATE DATABASE`. However, for a read-only database, the statement produced by `SHOW CREATE DATABASE` does include `READ ONLY=1` within a comment to indicate its read-only status:

```
mysql> ALTER DATABASE mydb READ ONLY = 1;
```

```
mysql> SHOW CREATE DATABASE mydb\G
*************************** 1. row ***************************
       Database: mydb
Create Database: CREATE DATABASE `mydb`
                 /*!40100 DEFAULT CHARACTER SET utf8mb4
                         COLLATE utf8mb4_0900_ai_ci */
                 /*!80016 DEFAULT ENCRYPTION='N' */
                 /* READ ONLY = 1 */
```

If the server executes a `CREATE DATABASE` statement containing such a comment, the server ignores the comment and the `READ ONLY` option is not processed. This has implications for `mysqldump` and `mysqlpump`, which use `SHOW CREATE DATABASE` to produce `CREATE DATABASE` statements in dump output:

- In a dump file, the `CREATE DATABASE` statement for a read-only database contains the commented `READ ONLY` option.
- The dump file can be restored as usual, but because the server ignores the commented `READ ONLY` option, the restored database is *not* read only. If the database is to be read only after being restored, you must execute `ALTER DATABASE` manually to make it so.

Suppose that `mydb` is read only and you dump it as follows:

```
$> mysqldump --databases mydb > mydb.sql
```

A restore operation later must be followed by `ALTER DATABASE` if `mydb` should still be read only:

```
$> mysql
mysql> SOURCE mydb.sql;
mysql> ALTER DATABASE mydb READ ONLY = 1;
```

MySQL Enterprise Backup is not subject to this issue. It backs up and restores a read-only database like any other, but enables the `READ ONLY` option at restore time if it was enabled at backup time.

`ALTER DATABASE` is written to the binary log, so a change to the `READ ONLY` option on a replication source server also affects replicas. To prevent this from happening, binary logging must be disabled prior to execution of the `ALTER DATABASE` statement. For example, to prepare for migrating a database without affecting replicas, perform these operations:

1. Within a single session, disable binary logging and enable `READ ONLY` for the database:

   ```
   mysql> SET sql_log_bin = OFF;
   mysql> ALTER DATABASE mydb READ ONLY = 1;
   ```

2. Dump the database, for example, with `mysqldump` or `mysqlpump`:

   ```
   $> mysqldump --databases mydb > mydb.sql
   ```

3. Within a single session, disable binary logging and disable `READ ONLY` for the database:

   ```
   mysql> SET sql_log_bin = OFF;
   mysql> ALTER DATABASE mydb READ ONLY = 0;
   ```

## 13.1.3 ALTER EVENT Statement

```
ALTER
    [DEFINER = user]
    EVENT event_name
    [ON SCHEDULE schedule]
    [ON COMPLETION [NOT] PRESERVE]
    [RENAME TO new_event_name]
    [ENABLE | DISABLE | DISABLE ON SLAVE]
    [COMMENT 'string']
    [DO event_body]
```

The `ALTER EVENT` statement changes one or more of the characteristics of an existing event without the need to drop and recreate it. The syntax for each of the `DEFINER`, `ON SCHEDULE`, `ON`

`COMPLETION`, `COMMENT`, `ENABLE` / `DISABLE`, and `DO` clauses is exactly the same as when used with `CREATE EVENT`. (See Section 13.1.13, "CREATE EVENT Statement".)

Any user can alter an event defined on a database for which that user has the `EVENT` privilege. When a user executes a successful `ALTER EVENT` statement, that user becomes the definer for the affected event.

`ALTER EVENT` works only with an existing event:

```
mysql> ALTER EVENT no_such_event
     >     ON SCHEDULE
     >       EVERY '2:3' DAY_HOUR;
ERROR 1517 (HY000): Unknown event 'no_such_event'
```

In each of the following examples, assume that the event named `myevent` is defined as shown here:

```
CREATE EVENT myevent
    ON SCHEDULE
      EVERY 6 HOUR
    COMMENT 'A sample comment.'
    DO
      UPDATE myschema.mytable SET mycol = mycol + 1;
```

The following statement changes the schedule for `myevent` from once every six hours starting immediately to once every twelve hours, starting four hours from the time the statement is run:

```
ALTER EVENT myevent
    ON SCHEDULE
      EVERY 12 HOUR
    STARTS CURRENT_TIMESTAMP + INTERVAL 4 HOUR;
```

It is possible to change multiple characteristics of an event in a single statement. This example changes the SQL statement executed by `myevent` to one that deletes all records from `mytable`; it also changes the schedule for the event such that it executes once, one day after this `ALTER EVENT` statement is run.

```
ALTER EVENT myevent
    ON SCHEDULE
      AT CURRENT_TIMESTAMP + INTERVAL 1 DAY
    DO
      TRUNCATE TABLE myschema.mytable;
```

Specify the options in an `ALTER EVENT` statement only for those characteristics that you want to change; omitted options keep their existing values. This includes any default values for `CREATE EVENT` such as `ENABLE`.

To disable `myevent`, use this `ALTER EVENT` statement:

```
ALTER EVENT myevent
    DISABLE;
```

The `ON SCHEDULE` clause may use expressions involving built-in MySQL functions and user variables to obtain any of the *`timestamp`* or *`interval`* values which it contains. You cannot use stored routines or loadable functions in such expressions, and you cannot use any table references; however, you can use `SELECT FROM DUAL`. This is true for both `ALTER EVENT` and `CREATE EVENT` statements. References to stored routines, loadable functions, and tables in such cases are specifically not permitted, and fail with an error (see Bug #22830).

Although an `ALTER EVENT` statement that contains another `ALTER EVENT` statement in its `DO` clause appears to succeed, when the server attempts to execute the resulting scheduled event, the execution fails with an error.

To rename an event, use the `ALTER EVENT` statement's `RENAME TO` clause. This statement renames the event `myevent` to `yourevent`:

```
ALTER EVENT myevent
```

```
    RENAME TO yourevent;
```

You can also move an event to a different database using `ALTER EVENT ... RENAME TO ...` and *`db_name.event_name`* notation, as shown here:

```
ALTER EVENT olddb.myevent
    RENAME TO newdb.myevent;
```

To execute the previous statement, the user executing it must have the `EVENT` privilege on both the `olddb` and `newdb` databases.

**Note**

There is no `RENAME EVENT` statement.

The value `DISABLE ON SLAVE` is used on a replica instead of `ENABLE` or `DISABLE` to indicate an event that was created on the replication source server and replicated to the replica, but that is not executed on the replica. Normally, `DISABLE ON SLAVE` is set automatically as required; however, there are some circumstances under which you may want or need to change it manually. See Section 17.5.1.16, “Replication of Invoked Features”, for more information.

## 13.1.4 ALTER FUNCTION Statement

```
ALTER FUNCTION func_name [characteristic ...]

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}
```

This statement can be used to change the characteristics of a stored function. More than one change may be specified in an `ALTER FUNCTION` statement. However, you cannot change the parameters or body of a stored function using this statement; to make such changes, you must drop and re-create the function using `DROP FUNCTION` and `CREATE FUNCTION`.

You must have the `ALTER ROUTINE` privilege for the function. (That privilege is granted automatically to the function creator.) If binary logging is enabled, the `ALTER FUNCTION` statement might also require the `SUPER` privilege, as described in Section 25.7, “Stored Program Binary Logging”.

## 13.1.5 ALTER INSTANCE Statement

```
ALTER INSTANCE instance_action

instance_action: {
  | {ENABLE|DISABLE} INNODB REDO_LOG
  | ROTATE INNODB MASTER KEY
  | ROTATE BINLOG MASTER KEY
  | RELOAD TLS
      [FOR CHANNEL {mysql_main | mysql_admin}]
      [NO ROLLBACK ON ERROR]
  | RELOAD KEYRING
}
```

`ALTER INSTANCE` defines actions applicable to a MySQL server instance. The statement supports these actions:

- `ALTER INSTANCE {ENABLE | DISABLE} INNODB REDO_LOG`

  This action enables or disables `InnoDB` redo logging. Redo logging is enabled by default. This feature is intended only for loading data into a new MySQL instance. The statement is not written to the binary log. This action was introduced in MySQL 8.0.21.

**Warning**

*Do not disable redo logging on a production system.* While it is permitted to shut down and restart the server while redo logging is disabled, an unexpected server stoppage while redo logging is disabled can cause data loss and instance corruption.

An `ALTER INSTANCE [ENABLE|DISABLE] INNODB REDO_LOG` operation requires an exclusive backup lock, which prevents other `ALTER INSTANCE` operations from executing concurrently. Other `ALTER INSTANCE` operations must wait for the lock to be released before executing.

For more information, see Disabling Redo Logging.

- `ALTER INSTANCE ROTATE INNODB MASTER KEY`

  This action rotates the master encryption key used for `InnoDB` tablespace encryption. Key rotation requires the `ENCRYPTION_KEY_ADMIN` or `SUPER` privilege. To perform this action, a keyring plugin must be installed and configured. For instructions, see Section 6.4.4, “The MySQL Keyring”.

  `ALTER INSTANCE ROTATE INNODB MASTER KEY` supports concurrent DML. However, it cannot be run concurrently with `CREATE TABLE ... ENCRYPTION` or `ALTER TABLE ... ENCRYPTION` operations, and locks are taken to prevent conflicts that could arise from concurrent execution of these statements. If one of the conflicting statements is running, it must complete before another can proceed.

  `ALTER INSTANCE ROTATE INNODB MASTER KEY` statements are written to the binary log so that they can be executed on replicated servers.

  For additional `ALTER INSTANCE ROTATE INNODB MASTER KEY` usage information, see Section 15.13, “InnoDB Data-at-Rest Encryption”.

- `ALTER INSTANCE ROTATE BINLOG MASTER KEY`

  This action rotates the binary log master key used for binary log encryption. Key rotation for the binary log master key requires the `BINLOG_ENCRYPTION_ADMIN` or `SUPER` privilege. The statement cannot be used if the `binlog_encryption` system variable is set to `OFF`. To perform this action, a keyring plugin must be installed and configured. For instructions, see Section 6.4.4, “The MySQL Keyring”.

  `ALTER INSTANCE ROTATE BINLOG MASTER KEY` actions are not written to the binary log and are not executed on replicas. Binary log master key rotation can therefore be carried out in replication environments including a mix of MySQL versions. To schedule regular rotation of the binary log master key on all applicable source and replica servers, you can enable the MySQL Event Scheduler on each server and issue the `ALTER INSTANCE ROTATE BINLOG MASTER KEY` statement using a `CREATE EVENT` statement. If you rotate the binary log master key because you suspect that the current or any of the previous binary log master keys might have been compromised, issue the statement on every applicable source and replica server, which enables you to verify immediate compliance.

  For additional `ALTER INSTANCE ROTATE BINLOG MASTER KEY` usage information, including what to do if the process does not complete correctly or is interrupted by an unexpected server halt, see Section 17.3.2, “Encrypting Binary Log Files and Relay Log Files”.

- `ALTER INSTANCE RELOAD TLS`

  This action reconfigures a TLS context from the current values of the system variables that define the context. It also updates the status variables that reflect the active context values. This action requires the `CONNECTION_ADMIN` privilege. For additional information about reconfiguring the TLS context, including which system and status variables are context-related, see Server-Side Runtime Configuration and Monitoring for Encrypted Connections.

By default, the statement reloads the TLS context for the main connection interface. If the `FOR CHANNEL` clause (available as of MySQL 8.0.21) is given, the statement reloads the TLS context for the named channel: `mysql_main` for the main connection interface, `mysql_admin` for the administrative connection interface. For information about the different interfaces, see Section 5.1.12.1, “Connection Interfaces”. The updated TLS context properties are exposed in the Performance Schema `tls_channel_status` table. See Section 27.12.21.8, “The tls_channel_status Table”.

Updating the TLS context for the main interface may also affect the administrative interface because unless some nondefault TLS value is configured for that interface, it uses the same TLS context as the main interface.

**Note**

When you reload the TLS context, OpenSSL reloads the file containing the CRL (certificate revocation list) as part of the process. If the CRL file is large, the server allocates a large chunk of memory (ten times the file size), which is doubled while the new instance is being loaded and the old one has not yet been released. The process resident memory is not immediately reduced after a large allocation is freed, so if you issue the `ALTER INSTANCE RELOAD TLS` statement repeatedly with a large CRL file, the process resident memory usage may grow as a result of this.

By default, the `RELOAD TLS` action rolls back with an error and has no effect if the configuration values do not permit creation of the new TLS context. The previous context values continue to be used for new connections. If the optional `NO ROLLBACK ON ERROR` clause is given and the new context cannot be created, rollback does not occur. Instead, a warning is generated and encryption is disabled for new connections on the interface to which the statement applies.

`ALTER INSTANCE RELOAD TLS` statements are not written to the binary log (and thus are not replicated). TLS configuration is local and depends on local files not necessarily present on all servers involved.

- `ALTER INSTANCE RELOAD KEYRING`

  If a keyring component is installed, this action tells the component to re-read its configuration file and reinitialize any keyring in-memory data. If you modify the component configuration at runtime, the new configuration does not take effect until you perform this action. Keyring reloading requires the `ENCRYPTION_KEY_ADMIN` privilege. This action was added in MySQL 8.0.24.

  This action enables reconfiguring only the currently installed keyring component. It does not enable changing which component is installed. For example, if you change the configuration for the installed keyring component, `ALTER INSTANCE RELOAD KEYRING` causes the new configuration to take effect. On the other hand, if you change the keyring component named in the server manifest file, `ALTER INSTANCE RELOAD KEYRING` has no effect and the current component remains installed.

  `ALTER INSTANCE RELOAD KEYRING` statements are not written to the binary log (and thus are not replicated).

## 13.1.6 ALTER LOGFILE GROUP Statement

```
ALTER LOGFILE GROUP logfile_group
    ADD UNDOFILE 'file_name'
    [INITIAL_SIZE [=] size]
    [WAIT]
    ENGINE [=] engine_name
```

This statement adds an `UNDO` file named '*`file_name`*' to an existing log file group *`logfile_group`*. An `ALTER LOGFILE GROUP` statement has one and only one `ADD UNDOFILE` clause. No `DROP UNDOFILE` clause is currently supported.

**Note**

All NDB Cluster Disk Data objects share the same namespace. This means that *each Disk Data object* must be uniquely named (and not merely each Disk Data object of a given type). For example, you cannot have a tablespace and an undo log file with the same name, or an undo log file and a data file with the same name.

The optional `INITIAL_SIZE` parameter sets the `UNDO` file's initial size in bytes; if not specified, the initial size defaults to 134217728 (128 MB). You may optionally follow *`size`* with a one-letter abbreviation for an order of magnitude, similar to those used in `my.cnf`. Generally, this is one of the letters `M` (megabytes) or `G` (gigabytes). (Bug #13116514, Bug #16104705, Bug #62858)

On 32-bit systems, the maximum supported value for `INITIAL_SIZE` is 4294967296 (4 GB). (Bug #29186)

The minimum allowed value for `INITIAL_SIZE` is 1048576 (1 MB). (Bug #29574)

**Note**

`WAIT` is parsed but otherwise ignored. This keyword currently has no effect, and is intended for future expansion.

The `ENGINE` parameter (required) determines the storage engine which is used by this log file group, with *`engine_name`* being the name of the storage engine. Currently, the only accepted values for *`engine_name`* are "`NDBCLUSTER`" and "`NDB`". The two values are equivalent.

Here is an example, which assumes that the log file group `lg_3` has already been created using `CREATE LOGFILE GROUP` (see Section 13.1.16, "CREATE LOGFILE GROUP Statement"):

```
ALTER LOGFILE GROUP lg_3
    ADD UNDOFILE 'undo_10.dat'
    INITIAL_SIZE=32M
    ENGINE=NDBCLUSTER;
```

When `ALTER LOGFILE GROUP` is used with `ENGINE = NDBCLUSTER` (alternatively, `ENGINE = NDB`), an `UNDO` log file is created on each NDB Cluster data node. You can verify that the `UNDO` files were created and obtain information about them by querying the Information Schema `FILES` table. For example:

```
mysql> SELECT FILE_NAME, LOGFILE_GROUP_NUMBER, EXTRA
    -> FROM INFORMATION_SCHEMA.FILES
    -> WHERE LOGFILE_GROUP_NAME = 'lg_3';
+-------------+--------------------+----------------+
| FILE_NAME   | LOGFILE_GROUP_NUMBER | EXTRA          |
+-------------+--------------------+----------------+
| newdata.dat |                  0 | CLUSTER_NODE=3 |
| newdata.dat |                  0 | CLUSTER_NODE=4 |
| undo_10.dat |                 11 | CLUSTER_NODE=3 |
| undo_10.dat |                 11 | CLUSTER_NODE=4 |
+-------------+--------------------+----------------+
4 rows in set (0.01 sec)
```

(See Section 26.3.15, "The INFORMATION_SCHEMA FILES Table".)

Memory used for `UNDO_BUFFER_SIZE` comes from the global pool whose size is determined by the value of the `SharedGlobalMemory` data node configuration parameter. This includes any default value implied for this option by the setting of the `InitialLogFileGroup` data node configuration parameter.

`ALTER LOGFILE GROUP` is useful only with Disk Data storage for NDB Cluster. For more information, see Section 23.6.11, "NDB Cluster Disk Data Tables".

## 13.1.7 ALTER PROCEDURE Statement

```
ALTER PROCEDURE proc_name [characteristic ...]
```

```
characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}
```

This statement can be used to change the characteristics of a stored procedure. More than one change may be specified in an `ALTER PROCEDURE` statement. However, you cannot change the parameters or body of a stored procedure using this statement; to make such changes, you must drop and re-create the procedure using `DROP PROCEDURE` and `CREATE PROCEDURE`.

You must have the `ALTER ROUTINE` privilege for the procedure. By default, that privilege is granted automatically to the procedure creator. This behavior can be changed by disabling the `automatic_sp_privileges` system variable. See Section 25.2.2, “Stored Routines and MySQL Privileges”.

## 13.1.8 ALTER SERVER Statement

```
ALTER SERVER  server_name
    OPTIONS (option [, option] ...)
```

Alters the server information for *`server_name`*, adjusting any of the options permitted in the `CREATE SERVER` statement. The corresponding fields in the `mysql.servers` table are updated accordingly. This statement requires the `SUPER` privilege.

For example, to update the `USER` option:

```
ALTER SERVER s OPTIONS (USER 'sally');
```

`ALTER SERVER` causes an implicit commit. See Section 13.3.3, “Statements That Cause an Implicit Commit”.

`ALTER SERVER` is not written to the binary log, regardless of the logging format that is in use.

## 13.1.9 ALTER TABLE Statement

```
ALTER TABLE tbl_name
    [alter_option [, alter_option] ...]
    [partition_options]

alter_option: {
    table_options
  | ADD [COLUMN] col_name column_definition
        [FIRST | AFTER col_name]
  | ADD [COLUMN] (col_name column_definition,...)
  | ADD {INDEX | KEY} [index_name]
        [index_type] (key_part,...) [index_option] ...
  | ADD {FULLTEXT | SPATIAL} [INDEX | KEY] [index_name]
        (key_part,...) [index_option] ...
  | ADD [CONSTRAINT [symbol]] PRIMARY KEY
        [index_type] (key_part,...)
        [index_option] ...
  | ADD [CONSTRAINT [symbol]] UNIQUE [INDEX | KEY]
        [index_name] [index_type] (key_part,...)
        [index_option] ...
  | ADD [CONSTRAINT [symbol]] FOREIGN KEY
        [index_name] (col_name,...)
        reference_definition
  | ADD [CONSTRAINT [symbol]] CHECK (expr) [[NOT] ENFORCED]
  | DROP {CHECK | CONSTRAINT} symbol
  | ALTER {CHECK | CONSTRAINT} symbol [NOT] ENFORCED
  | ALGORITHM [=] {DEFAULT | INSTANT | INPLACE | COPY}
  | ALTER [COLUMN] col_name {
        SET DEFAULT {literal | (expr)}
      | SET {VISIBLE | INVISIBLE}
      | DROP DEFAULT
    }
```

```
  | ALTER INDEX index_name {VISIBLE | INVISIBLE}
  | CHANGE [COLUMN] old_col_name new_col_name column_definition
        [FIRST | AFTER col_name]
  | [DEFAULT] CHARACTER SET [=] charset_name [COLLATE [=] collation_name]
  | CONVERT TO CHARACTER SET charset_name [COLLATE collation_name]
  | {DISABLE | ENABLE} KEYS
  | {DISCARD | IMPORT} TABLESPACE
  | DROP [COLUMN] col_name
  | DROP {INDEX | KEY} index_name
  | DROP PRIMARY KEY
  | DROP FOREIGN KEY fk_symbol
  | FORCE
  | LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE}
  | MODIFY [COLUMN] col_name column_definition
        [FIRST | AFTER col_name]
  | ORDER BY col_name [, col_name] ...
  | RENAME COLUMN old_col_name TO new_col_name
  | RENAME {INDEX | KEY} old_index_name TO new_index_name
  | RENAME [TO | AS] new_tbl_name
  | {WITHOUT | WITH} VALIDATION
}

partition_options:
    partition_option [partition_option] ...

partition_option: {
    ADD PARTITION (partition_definition)
  | DROP PARTITION partition_names
  | DISCARD PARTITION {partition_names | ALL} TABLESPACE
  | IMPORT PARTITION {partition_names | ALL} TABLESPACE
  | TRUNCATE PARTITION {partition_names | ALL}
  | COALESCE PARTITION number
  | REORGANIZE PARTITION partition_names INTO (partition_definitions)
  | EXCHANGE PARTITION partition_name WITH TABLE tbl_name [{WITH | WITHOUT} VALIDATION]
  | ANALYZE PARTITION {partition_names | ALL}
  | CHECK PARTITION {partition_names | ALL}
  | OPTIMIZE PARTITION {partition_names | ALL}
  | REBUILD PARTITION {partition_names | ALL}
  | REPAIR PARTITION {partition_names | ALL}
  | REMOVE PARTITIONING
}

key_part: {col_name [(length)] | (expr)} [ASC | DESC]

index_type:
    USING {BTREE | HASH}

index_option: {
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
  | {VISIBLE | INVISIBLE}
}

table_options:
    table_option [[,] table_option] ...

table_option: {
    AUTOEXTEND_SIZE [=] value
  | AUTO_INCREMENT [=] value
  | AVG_ROW_LENGTH [=] value
  | [DEFAULT] CHARACTER SET [=] charset_name
  | CHECKSUM [=] {0 | 1}
  | [DEFAULT] COLLATE [=] collation_name
  | COMMENT [=] 'string'
  | COMPRESSION [=] {'ZLIB' | 'LZ4' | 'NONE'}
  | CONNECTION [=] 'connect_string'
  | {DATA | INDEX} DIRECTORY [=] 'absolute path to directory'
  | DELAY_KEY_WRITE [=] {0 | 1}
  | ENCRYPTION [=] {'Y' | 'N'}
  | ENGINE [=] engine_name
```

```
  | ENGINE_ATTRIBUTE [=] 'string'
  | INSERT_METHOD [=] { NO | FIRST | LAST }
  | KEY_BLOCK_SIZE [=] value
  | MAX_ROWS [=] value
  | MIN_ROWS [=] value
  | PACK_KEYS [=] {0 | 1 | DEFAULT}
  | PASSWORD [=] 'string'
  | ROW_FORMAT [=] {DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUNDANT | COMPACT}
  | SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
  | STATS_AUTO_RECALC [=] {DEFAULT | 0 | 1}
  | STATS_PERSISTENT [=] {DEFAULT | 0 | 1}
  | STATS_SAMPLE_PAGES [=] value
  | TABLESPACE tablespace_name [STORAGE {DISK | MEMORY}]
  | UNION [=] (tbl_name[,tbl_name]...)
}

partition_options:
    (see CREATE TABLE options)
```

`ALTER TABLE` changes the structure of a table. For example, you can add or delete columns, create or destroy indexes, change the type of existing columns, or rename columns or the table itself. You can also change characteristics such as the storage engine used for the table or the table comment.

- To use `ALTER TABLE`, you need `ALTER`, `CREATE`, and `INSERT` privileges for the table. Renaming a table requires `ALTER` and `DROP` on the old table, `ALTER`, `CREATE`, and `INSERT` on the new table.
- Following the table name, specify the alterations to be made. If none are given, `ALTER TABLE` does nothing.
- The syntax for many of the permissible alterations is similar to clauses of the `CREATE TABLE` statement. *`column_definition`* clauses use the same syntax for `ADD` and `CHANGE` as for `CREATE TABLE`. For more information, see Section 13.1.20, “CREATE TABLE Statement”.
- The word `COLUMN` is optional and can be omitted, except for `RENAME COLUMN` (to distinguish a column-renaming operation from the `RENAME` table-renaming operation).
- Multiple `ADD`, `ALTER`, `DROP`, and `CHANGE` clauses are permitted in a single `ALTER TABLE` statement, separated by commas. This is a MySQL extension to standard SQL, which permits only one of each clause per `ALTER TABLE` statement. For example, to drop multiple columns in a single statement, do this:

  ```
  ALTER TABLE t2 DROP COLUMN c, DROP COLUMN d;
  ```

- If a storage engine does not support an attempted `ALTER TABLE` operation, a warning may result. Such warnings can be displayed with `SHOW WARNINGS`. See Section 13.7.7.42, “SHOW WARNINGS Statement”. For information on troubleshooting `ALTER TABLE`, see Section B.3.6.1, “Problems with ALTER TABLE”.
- For information about generated columns, see Section 13.1.9.2, “ALTER TABLE and Generated Columns”.
- For usage examples, see Section 13.1.9.3, “ALTER TABLE Examples”.
- `InnoDB` in MySQL 8.0.17 and later supports addition of multi-valued indexes on JSON columns using a *`key_part`* specification can take the form `(CAST json_path AS type ARRAY)`. See Multi-Valued Indexes, for detailed information regarding multi-valued index creation and usage of, as well as restrictions and limitations on multi-valued indexes.
- With the `mysql_info()` C API function, you can find out how many rows were copied by `ALTER TABLE`. See mysql_info().

There are several additional aspects to the `ALTER TABLE` statement, described under the following topics in this section:

- Table Options

- Performance and Space Requirements
- Concurrency Control
- Adding and Dropping Columns
- Renaming, Redefining, and Reordering Columns
- Primary Keys and Indexes
- Foreign Keys and Other Constraints
- Changing the Character Set
- Importing InnoDB Tables
- Row Order for MyISAM Tables
- Partitioning Options

## Table Options

*`table_options`* signifies table options of the kind that can be used in the `CREATE TABLE` statement, such as `ENGINE`, `AUTO_INCREMENT`, `AVG_ROW_LENGTH`, `MAX_ROWS`, `ROW_FORMAT`, or `TABLESPACE`.

For descriptions of all table options, see Section 13.1.20, “CREATE TABLE Statement”. However, `ALTER TABLE` ignores `DATA DIRECTORY` and `INDEX DIRECTORY` when given as table options. `ALTER TABLE` permits them only as partitioning options, and requires that you have the `FILE` privilege.

Use of table options with `ALTER TABLE` provides a convenient way of altering single table characteristics. For example:

- If `t1` is currently not an `InnoDB` table, this statement changes its storage engine to `InnoDB`:

```
ALTER TABLE t1 ENGINE = InnoDB;
```

  - See Section 15.6.1.5, “Converting Tables from MyISAM to InnoDB” for considerations when switching tables to the `InnoDB` storage engine.
  - When you specify an `ENGINE` clause, `ALTER TABLE` rebuilds the table. This is true even if the table already has the specified storage engine.
  - Running `ALTER TABLE *tbl_name* ENGINE=INNODB` on an existing `InnoDB` table performs a “null” `ALTER TABLE` operation, which can be used to defragment an `InnoDB` table, as described in Section 15.11.4, “Defragmenting a Table”. Running `ALTER TABLE *tbl_name* FORCE` on an `InnoDB` table performs the same function.
  - `ALTER TABLE *tbl_name* ENGINE=INNODB` and `ALTER TABLE *tbl_name* FORCE` use online DDL. For more information, see Section 15.12, “InnoDB and Online DDL”.
  - The outcome of attempting to change the storage engine of a table is affected by whether the desired storage engine is available and the setting of the `NO_ENGINE_SUBSTITUTION` SQL mode, as described in Section 5.1.11, “Server SQL Modes”.
  - To prevent inadvertent loss of data, `ALTER TABLE` cannot be used to change the storage engine of a table to `MERGE` or `BLACKHOLE`.

- To change the `InnoDB` table to use compressed row-storage format:

```
ALTER TABLE t1 ROW_FORMAT = COMPRESSED;
```

- The `ENCRYPTION` clause enables or disables page-level data encryption for an `InnoDB` table. A keyring plugin must be installed and configured to enable encryption.

If the `table_encryption_privilege_check` variable is enabled, the `TABLE_ENCRYPTION_ADMIN` privilege is required to use an `ENCRYPTION` clause with a setting that differs from the default schema encryption setting.

Prior to MySQL 8.0.16, the `ENCRYPTION` clause was only supported when altering tables residing in file-per-table tablespaces. As of MySQL 8.0.16, the `ENCRYPTION` clause is also supported for tables residing in general tablespaces.

For tables that reside in general tablespaces, table and tablespace encryption must match.

Altering table encryption by moving a table to a different tablespace or changing the storage engine is not permitted without explicitly specifying an `ENCRYPTION` clause.

As of MySQL 8.0.16, specifying an `ENCRYPTION` clause with a value other than `'N'` or `''` is not permitted if the table uses a storage engine that does not support encryption. Previously, the clause was accepted. Attempting to create a table without an `ENCRYPTION` clause in an encryption-enabled schema using a storage engine that does not support encryption is also not permitted.

For more information, see Section 15.13, “InnoDB Data-at-Rest Encryption”.

- To reset the current auto-increment value:

```
ALTER TABLE t1 AUTO_INCREMENT = 13;
```

You cannot reset the counter to a value less than or equal to the value that is currently in use. For both `InnoDB` and `MyISAM`, if the value is less than or equal to the maximum value currently in the `AUTO_INCREMENT` column, the value is reset to the current maximum `AUTO_INCREMENT` column value plus one.

- To change the default table character set:

```
ALTER TABLE t1 CHARACTER SET = utf8mb4;
```

See also Changing the Character Set.

- To add (or change) a table comment:

```
ALTER TABLE t1 COMMENT = 'New table comment';
```

- Use `ALTER TABLE` with the `TABLESPACE` option to move `InnoDB` tables between existing general tablespaces, file-per-table tablespaces, and the system tablespace. See Moving Tables Between Tablespaces Using ALTER TABLE.

  - `ALTER TABLE ... TABLESPACE` operations always cause a full table rebuild, even if the `TABLESPACE` attribute has not changed from its previous value.

  - `ALTER TABLE ... TABLESPACE` syntax does not support moving a table from a temporary tablespace to a persistent tablespace.

  - The `DATA DIRECTORY` clause, which is supported with `CREATE TABLE ... TABLESPACE`, is not supported with `ALTER TABLE ... TABLESPACE`, and is ignored if specified.

  - For more information about the capabilities and limitations of the `TABLESPACE` option, see `CREATE TABLE`.

- MySQL NDB Cluster 8.0 supports setting `NDB_TABLE` options for controlling a table's partition balance (fragment count type), read-from-any-replica capability, full replication, or any combination

of these, as part of the table comment for an `ALTER TABLE` statement in the same manner as for `CREATE TABLE`, as shown in this example:

```
ALTER TABLE t1 COMMENT = "NDB_TABLE=READ_BACKUP=0,PARTITION_BALANCE=FOR_RA_BY_NODE";
```

It is also possible to set `NDB_COMMENT` options for columns of `NDB` tables as part of an `ALTER TABLE` statement, like this one:

```
ALTER TABLE t1
  CHANGE COLUMN c1 c1 BLOB
    COMMENT = 'NDB_COLUMN=BLOB_INLINE_SIZE=4096,MAX_BLOB_PART_SIZE';
```

Setting the blob inline size in this fashion is supported by NDB 8.0.30 and later. Bear in mind that `ALTER TABLE ... COMMENT ...` discards any existing comment for the table. See Setting NDB_TABLE options, for additional information and examples.

- `ENGINE_ATTRIBUTE` and `SECONDARY_ENGINE_ATTRIBUTE` options (available as of MySQL 8.0.21) are used to specify table, column, and index attributes for primary and secondary storage engines. The options are reserved for future use. Index attributes cannot be altered. An index must be dropped and added back with the desired change, which can be performed in a single `ALTER TABLE` statement.

To verify that the table options were changed as intended, use `SHOW CREATE TABLE`, or query the Information Schema `TABLES` table.

## Performance and Space Requirements

`ALTER TABLE` operations are processed using one of the following algorithms:

- `COPY`: Operations are performed on a copy of the original table, and table data is copied from the original table to the new table row by row. Concurrent DML is not permitted.
- `INPLACE`: Operations avoid copying table data but may rebuild the table in place. An exclusive metadata lock on the table may be taken briefly during preparation and execution phases of the operation. Typically, concurrent DML is supported.
- `INSTANT`: Operations only modify metadata in the data dictionary. An exclusive metadata lock on the table may be taken briefly during the execution phase of the operation. Table data is unaffected, making operations instantaneous. Concurrent DML is permitted. (Introduced in MySQL 8.0.12)

For tables using the `NDB` storage engine, these algorithms work as follows:

- `COPY`: `NDB` creates a copy of the table and alters it; the NDB Cluster handler then copies the data between the old and new versions of the table. Subsequently, `NDB` deletes the old table and renames the new one.

  This is sometimes also referred to as a “copying” or “offline” `ALTER TABLE`.
- `INPLACE`: The data nodes make the required changes; the NDB Cluster handler does not copy data or otherwise take part.

  This is sometimes also referred to as a “non-copying” or “online” `ALTER TABLE`.
- `INSTANT`: Not supported by `NDB`.

See Section 23.6.12, “Online Operations with ALTER TABLE in NDB Cluster”, for more information.

The `ALGORITHM` clause is optional. If the `ALGORITHM` clause is omitted, MySQL uses `ALGORITHM=INSTANT` for storage engines and `ALTER TABLE` clauses that support it. Otherwise, `ALGORITHM=INPLACE` is used. If `ALGORITHM=INPLACE` is not supported, `ALGORITHM=COPY` is used.

**Note**

After adding a column to a partitioned table using `ALGORITHM=INSTANT`, it is no longer possible to perform `ALTER TABLE ... EXCHANGE PARTITION` on the table.

Specifying an `ALGORITHM` clause requires the operation to use the specified algorithm for clauses and storage engines that support it, or fail with an error otherwise. Specifying `ALGORITHM=DEFAULT` is the same as omitting the `ALGORITHM` clause.

`ALTER TABLE` operations that use the `COPY` algorithm wait for other operations that are modifying the table to complete. After alterations are applied to the table copy, data is copied over, the original table is deleted, and the table copy is renamed to the name of the original table. While the `ALTER TABLE` operation executes, the original table is readable by other sessions (with the exception noted shortly). Updates and writes to the table started after the `ALTER TABLE` operation begins are stalled until the new table is ready, then are automatically redirected to the new table. The temporary copy of the table is created in the database directory of the original table unless it is a `RENAME TO` operation that moves the table to a database that resides in a different directory.

The exception referred to earlier is that `ALTER TABLE` blocks reads (not just writes) at the point where it is ready to clear outdated table structures from the table and table definition caches. At this point, it must acquire an exclusive lock. To do so, it waits for current readers to finish, and blocks new reads and writes.

An `ALTER TABLE` operation that uses the `COPY` algorithm prevents concurrent DML operations. Concurrent queries are still allowed. That is, a table-copying operation always includes at least the concurrency restrictions of `LOCK=SHARED` (allow queries but not DML). You can further restrict concurrency for operations that support the `LOCK` clause by specifying `LOCK=EXCLUSIVE`, which prevents DML and queries. For more information, see Concurrency Control.

To force use of the `COPY` algorithm for an `ALTER TABLE` operation that would otherwise not use it, specify `ALGORITHM=COPY` or enable the `old_alter_table` system variable. If there is a conflict between the `old_alter_table` setting and an `ALGORITHM` clause with a value other than `DEFAULT`, the `ALGORITHM` clause takes precedence.

For `InnoDB` tables, an `ALTER TABLE` operation that uses the `COPY` algorithm on a table that resides in a shared tablespace can increase the amount of space used by the tablespace. Such operations require as much additional space as the data in the table plus indexes. For a table residing in a shared tablespace, the additional space used during the operation is not released back to the operating system as it is for a table that resides in a file-per-table tablespace.

For information about space requirements for online DDL operations, see Section 15.12.3, “Online DDL Space Requirements”.

`ALTER TABLE` operations that support the `INPLACE` algorithm include:

- `ALTER TABLE` operations supported by the `InnoDB` online DDL feature. See Section 15.12.1, “Online DDL Operations”.
- Renaming a table. MySQL renames files that correspond to the table *`tbl_name`* without making a copy. (You can also use the `RENAME TABLE` statement to rename tables. See Section 13.1.36, “RENAME TABLE Statement”.) Privileges granted specifically for the renamed table are not migrated to the new name. They must be changed manually.
- Operations that modify table metadata only. These operations are immediate because the server does not touch table contents. Metadata-only operations include:
  - Renaming a column. In NDB Cluster 8.0.18 and later, this operation can also be performed online.
  - Changing the default value of a column (except for `NDB` tables).

- Modifying the definition of an `ENUM` or `SET` column by adding new enumeration or set members to the *end* of the list of valid member values, as long as the storage size of the data type does not change. For example, adding a member to a `SET` column that has 8 members changes the required storage per value from 1 byte to 2 bytes; this requires a table copy. Adding members in the middle of the list causes renumbering of existing members, which requires a table copy.
- Changing the definition of a spatial column to remove the `SRID` attribute. (Adding or changing an `SRID` attribute requires a rebuild, and cannot be done in place, because the server must verify that all values have the specified `SRID` value.)
- As of MySQL 8.0.14, changing a column character set, when these conditions apply:
  - The column data type is `CHAR`, `VARCHAR`, a `TEXT` type, or `ENUM`.
  - The character set change is from `utf8mb3` to `utf8mb4`, or any character set to `binary`.
  - There is no index on the column.
- As of MySQL 8.0.14, changing a generated column, when these conditions apply:
  - For `InnoDB` tables, statements that modify generated stored columns but do not change their type, expression, or nullability.
  - For non-`InnoDB` tables, statements that modify generated stored or virtual columns but do not change their type, expression, or nullability.

  An example of such a change is a change to the column comment.

- Renaming an index.
- Adding or dropping a secondary index, for `InnoDB` and `NDB` tables. See Section 15.12.1, “Online DDL Operations”.
- For `NDB` tables, operations that add and drop indexes on variable-width columns. These operations occur online, without table copying and without blocking concurrent DML actions for most of their duration. See Section 23.6.12, “Online Operations with ALTER TABLE in NDB Cluster”.
- Modifying index visibility with an `ALTER INDEX` operation.
- Column modifications of tables containing generated columns that depend on columns with a `DEFAULT` value if the modified columns are not involved in the generated column expressions. For example, changing the `NULL` property of a separate column can be done in place without a table rebuild.

`ALTER TABLE` operations that support the `INSTANT` algorithm include:

- Adding a column. This feature is referred to as “Instant `ADD COLUMN`”. Limitations apply. See Section 15.12.1, “Online DDL Operations”.
- Dropping a column. This feature is referred to as “Instant `DROP COLUMN`”. Limitations apply. See Section 15.12.1, “Online DDL Operations”.
- Adding or dropping a virtual column.
- Adding or dropping a column default value.
- Modifying the definition of an `ENUM` or `SET` column. The same restrictions apply as described above for `ALGORITHM=INSTANT`.
- Changing the index type.
- Renaming a table. The same restrictions apply as described above for `ALGORITHM=INSTANT`.

For more information about operations that support ALGORITHM=INSTANT, see Section 15.12.1, “Online DDL Operations”.

ALTER TABLE upgrades MySQL 5.5 temporal columns to 5.6 format for ADD COLUMN, CHANGE COLUMN, MODIFY COLUMN, ADD INDEX, and FORCE operations. This conversion cannot be done using the INPLACE algorithm because the table must be rebuilt, so specifying ALGORITHM=INPLACE in these cases results in an error. Specify ALGORITHM=COPY if necessary.

If an ALTER TABLE operation on a multicolumn index used to partition a table by KEY changes the order of the columns, it can only be performed using ALGORITHM=COPY.

The WITHOUT VALIDATION and WITH VALIDATION clauses affect whether ALTER TABLE performs an in-place operation for virtual generated column modifications. See Section 13.1.9.2, “ALTER TABLE and Generated Columns”.

NDB Cluster 8.0 supports online operations using the same ALGORITHM=INPLACE syntax used with the standard MySQL Server. NDB does not support changing a tablespace online; beginning with NDB 8.0.21, it is disallowed. See Section 23.6.12, “Online Operations with ALTER TABLE in NDB Cluster”, for more information.

NDB 8.0.27 and later, when performing a copying ALTER TABLE, checks to ensure that no concurrent writes have been made to the affected table. If it finds that any have been made, NDB rejects the ALTER TABLE statement and raises ER_TABLE_DEF_CHANGED.

ALTER TABLE with DISCARD ... PARTITION ... TABLESPACE or IMPORT ... PARTITION ... TABLESPACE does not create any temporary tables or temporary partition files.

ALTER TABLE with ADD PARTITION, DROP PARTITION, COALESCE PARTITION, REBUILD PARTITION, or REORGANIZE PARTITION does not create temporary tables (except when used with NDB tables); however, these operations can and do create temporary partition files.

ADD or DROP operations for RANGE or LIST partitions are immediate operations or nearly so. ADD or COALESCE operations for HASH or KEY partitions copy data between all partitions, unless LINEAR HASH or LINEAR KEY was used; this is effectively the same as creating a new table, although the ADD or COALESCE operation is performed partition by partition. REORGANIZE operations copy only changed partitions and do not touch unchanged ones.

For MyISAM tables, you can speed up index re-creation (the slowest part of the alteration process) by setting the myisam_sort_buffer_size system variable to a high value.

## Concurrency Control

For ALTER TABLE operations that support it, you can use the LOCK clause to control the level of concurrent reads and writes on a table while it is being altered. Specifying a non-default value for this clause enables you to require a certain amount of concurrent access or exclusivity during the alter operation, and halts the operation if the requested degree of locking is not available.

Only LOCK = DEFAULT is permitted for operations that use ALGORITHM=INSTANT. The other LOCK clause parameters are not applicable.

The parameters for the LOCK clause are:

- LOCK = DEFAULT

  Maximum level of concurrency for the given ALGORITHM clause (if any) and ALTER TABLE operation: Permit concurrent reads and writes if supported. If not, permit concurrent reads if supported. If not, enforce exclusive access.

- LOCK = NONE

  If supported, permit concurrent reads and writes. Otherwise, an error occurs.

- `LOCK = SHARED`

  If supported, permit concurrent reads but block writes. Writes are blocked even if concurrent writes are supported by the storage engine for the given `ALGORITHM` clause (if any) and `ALTER TABLE` operation. If concurrent reads are not supported, an error occurs.

- `LOCK = EXCLUSIVE`

  Enforce exclusive access. This is done even if concurrent reads/writes are supported by the storage engine for the given `ALGORITHM` clause (if any) and `ALTER TABLE` operation.

## Adding and Dropping Columns

Use `ADD` to add new columns to a table, and `DROP` to remove existing columns. `DROP col_name` is a MySQL extension to standard SQL.

To add a column at a specific position within a table row, use `FIRST` or `AFTER col_name`. The default is to add the column last.

If a table contains only one column, the column cannot be dropped. If what you intend is to remove the table, use the `DROP TABLE` statement instead.

If columns are dropped from a table, the columns are also removed from any index of which they are a part. If all columns that make up an index are dropped, the index is dropped as well. If you use `CHANGE` or `MODIFY` to shorten a column for which an index exists on the column, and the resulting column length is less than the index length, MySQL shortens the index automatically.

For `ALTER TABLE ... ADD`, if the column has an expression default value that uses a nondeterministic function, the statement may produce a warning or error. For further information, see Section 11.6, “Data Type Default Values”, and Section 17.1.3.7, “Restrictions on Replication with GTIDs”.

## Renaming, Redefining, and Reordering Columns

The `CHANGE`, `MODIFY`, `RENAME COLUMN`, and `ALTER` clauses enable the names and definitions of existing columns to be altered. They have these comparative characteristics:

- `CHANGE`:
  - Can rename a column and change its definition, or both.
  - Has more capability than `MODIFY` or `RENAME COLUMN`, but at the expense of convenience for some operations. `CHANGE` requires naming the column twice if not renaming it, and requires respecifying the column definition if only renaming it.
  - With `FIRST` or `AFTER`, can reorder columns.
- `MODIFY`:
  - Can change a column definition but not its name.
  - More convenient than `CHANGE` to change a column definition without renaming it.
  - With `FIRST` or `AFTER`, can reorder columns.
- `RENAME COLUMN`:
  - Can change a column name but not its definition.
  - More convenient than `CHANGE` to rename a column without changing its definition.
- `ALTER`: Used only to change a column default value.

`CHANGE` is a MySQL extension to standard SQL. `MODIFY` and `RENAME COLUMN` are MySQL extensions for Oracle compatibility.

To alter a column to change both its name and definition, use `CHANGE`, specifying the old and new names and the new definition. For example, to rename an `INT NOT NULL` column from `a` to `b` and change its definition to use the `BIGINT` data type while retaining the `NOT NULL` attribute, do this:

```
ALTER TABLE t1 CHANGE a b BIGINT NOT NULL;
```

To change a column definition but not its name, use `CHANGE` or `MODIFY`. With `CHANGE`, the syntax requires two column names, so you must specify the same name twice to leave the name unchanged. For example, to change the definition of column `b`, do this:

```
ALTER TABLE t1 CHANGE b b INT NOT NULL;
```

`MODIFY` is more convenient to change the definition without changing the name because it requires the column name only once:

```
ALTER TABLE t1 MODIFY b INT NOT NULL;
```

To change a column name but not its definition, use `CHANGE` or `RENAME COLUMN`. With `CHANGE`, the syntax requires a column definition, so to leave the definition unchanged, you must respecify the definition the column currently has. For example, to rename an `INT NOT NULL` column from `b` to `a`, do this:

```
ALTER TABLE t1 CHANGE b a INT NOT NULL;
```

`RENAME COLUMN` is more convenient to change the name without changing the definition because it requires only the old and new names:

```
ALTER TABLE t1 RENAME COLUMN b TO a;
```

In general, you cannot rename a column to a name that already exists in the table. However, this is sometimes not the case, such as when you swap names or move them through a cycle. If a table has columns named `a`, `b`, and `c`, these are valid operations:

```
-- swap a and b
ALTER TABLE t1 RENAME COLUMN a TO b,
               RENAME COLUMN b TO a;
-- "rotate" a, b, c through a cycle
ALTER TABLE t1 RENAME COLUMN a TO b,
               RENAME COLUMN b TO c,
               RENAME COLUMN c TO a;
```

For column definition changes using `CHANGE` or `MODIFY`, the definition must include the data type and all attributes that should apply to the new column, other than index attributes such as `PRIMARY KEY` or `UNIQUE`. Attributes present in the original definition but not specified for the new definition are not carried forward. Suppose that a column `col1` is defined as `INT UNSIGNED DEFAULT 1 COMMENT 'my column'` and you modify the column as follows, intending to change only `INT` to `BIGINT`:

```
ALTER TABLE t1 MODIFY col1 BIGINT;
```

That statement changes the data type from `INT` to `BIGINT`, but it also drops the `UNSIGNED`, `DEFAULT`, and `COMMENT` attributes. To retain them, the statement must include them explicitly:

```
ALTER TABLE t1 MODIFY col1 BIGINT UNSIGNED DEFAULT 1 COMMENT 'my column';
```

For data type changes using `CHANGE` or `MODIFY`, MySQL tries to convert existing column values to the new type as well as possible.

**Warning**

This conversion may result in alteration of data. For example, if you shorten a string column, values may be truncated. To prevent the operation from succeeding if conversions to the new data type would result in loss of data,

enable strict SQL mode before using `ALTER TABLE` (see Section 5.1.11, “Server SQL Modes”).

If you use `CHANGE` or `MODIFY` to shorten a column for which an index exists on the column, and the resulting column length is less than the index length, MySQL shortens the index automatically.

For columns renamed by `CHANGE` or `RENAME COLUMN`, MySQL automatically renames these references to the renamed column:

- Indexes that refer to the old column, including invisible indexes and disabled `MyISAM` indexes.
- Foreign keys that refer to the old column.

For columns renamed by `CHANGE` or `RENAME COLUMN`, MySQL does not automatically rename these references to the renamed column:

- Generated column and partition expressions that refer to the renamed column. You must use `CHANGE` to redefine such expressions in the same `ALTER TABLE` statement as the one that renames the column.
- Views and stored programs that refer to the renamed column. You must manually alter the definition of these objects to refer to the new column name.

To reorder columns within a table, use `FIRST` and `AFTER` in `CHANGE` or `MODIFY` operations.

`ALTER ... SET DEFAULT` or `ALTER ... DROP DEFAULT` specify a new default value for a column or remove the old default value, respectively. If the old default is removed and the column can be `NULL`, the new default is `NULL`. If the column cannot be `NULL`, MySQL assigns a default value as described in Section 11.6, “Data Type Default Values”.

As of MySQL 8.0.23, `ALTER ... SET VISIBLE` and `ALTER ... SET INVISIBLE` enable column visibility to be changed. See Section 13.1.20.10, “Invisible Columns”.

## Primary Keys and Indexes

`DROP PRIMARY KEY` drops the primary key. If there is no primary key, an error occurs. For information about the performance characteristics of primary keys, especially for `InnoDB` tables, see Section 8.3.2, “Primary Key Optimization”.

If the `sql_require_primary_key` system variable is enabled, attempting to drop a primary key produces an error.

If you add a `UNIQUE INDEX` or `PRIMARY KEY` to a table, MySQL stores it before any nonunique index to permit detection of duplicate keys as early as possible.

`DROP INDEX` removes an index. This is a MySQL extension to standard SQL. See Section 13.1.27, “DROP INDEX Statement”. To determine index names, use `SHOW INDEX FROM` *`tbl_name`*.

Some storage engines permit you to specify an index type when creating an index. The syntax for the *`index_type`* specifier is `USING` *`type_name`*. For details about `USING`, see Section 13.1.15, “CREATE INDEX Statement”. The preferred position is after the column list. Expect support for use of the option before the column list to be removed in a future MySQL release.

*`index_option`* values specify additional options for an index. `USING` is one such option. For details about permissible *`index_option`* values, see Section 13.1.15, “CREATE INDEX Statement”.

`RENAME INDEX` *`old_index_name`* `TO` *`new_index_name`* renames an index. This is a MySQL extension to standard SQL. The content of the table remains unchanged. *`old_index_name`* must be the name of an existing index in the table that is not dropped by the same `ALTER TABLE` statement. *`new_index_name`* is the new index name, which cannot duplicate the name of an index in the resulting table after changes have been applied. Neither index name can be `PRIMARY`.

If you use `ALTER TABLE` on a `MyISAM` table, all nonunique indexes are created in a separate batch (as for `REPAIR TABLE`). This should make `ALTER TABLE` much faster when you have many indexes.

For MyISAM tables, key updating can be controlled explicitly. Use ALTER TABLE ... DISABLE KEYS to tell MySQL to stop updating nonunique indexes. Then use ALTER TABLE ... ENABLE KEYS to re-create missing indexes. MyISAM does this with a special algorithm that is much faster than inserting keys one by one, so disabling keys before performing bulk insert operations should give a considerable speedup. Using ALTER TABLE ... DISABLE KEYS requires the INDEX privilege in addition to the privileges mentioned earlier.

While the nonunique indexes are disabled, they are ignored for statements such as SELECT and EXPLAIN that otherwise would use them.

After an ALTER TABLE statement, it may be necessary to run ANALYZE TABLE to update index cardinality information. See Section 13.7.7.22, “SHOW INDEX Statement”.

The ALTER INDEX operation permits an index to be made visible or invisible. An invisible index is not used by the optimizer. Modification of index visibility applies to indexes other than primary keys (either explicit or implicit). This feature is storage engine neutral (supported for any engine). For more information, see Section 8.3.12, “Invisible Indexes”.

## Foreign Keys and Other Constraints

The FOREIGN KEY and REFERENCES clauses are supported by the InnoDB and NDB storage engines, which implement ADD [CONSTRAINT [*symbol*]] FOREIGN KEY [*index_name*] (...) REFERENCES ... (...). See Section 13.1.20.5, “FOREIGN KEY Constraints”. For other storage engines, the clauses are parsed but ignored.

For ALTER TABLE, unlike CREATE TABLE, ADD FOREIGN KEY ignores *index_name* if given and uses an automatically generated foreign key name. As a workaround, include the CONSTRAINT clause to specify the foreign key name:

```
ADD CONSTRAINT name FOREIGN KEY (....) ...
```

**Important**

MySQL silently ignores inline REFERENCES specifications, where the references are defined as part of the column specification. MySQL accepts only REFERENCES clauses defined as part of a separate FOREIGN KEY specification.

**Note**

Partitioned InnoDB tables do not support foreign keys. This restriction does not apply to NDB tables, including those explicitly partitioned by [LINEAR] KEY. For more information, see Section 24.6.2, “Partitioning Limitations Relating to Storage Engines”.

MySQL Server and NDB Cluster both support the use of ALTER TABLE to drop foreign keys:

```
ALTER TABLE tbl_name DROP FOREIGN KEY fk_symbol;
```

Adding and dropping a foreign key in the same ALTER TABLE statement is supported for ALTER TABLE ... ALGORITHM=INPLACE but not for ALTER TABLE ... ALGORITHM=COPY.

The server prohibits changes to foreign key columns that have the potential to cause loss of referential integrity. A workaround is to use ALTER TABLE ... DROP FOREIGN KEY before changing the column definition and ALTER TABLE ... ADD FOREIGN KEY afterward. Examples of prohibited changes include:

- Changes to the data type of foreign key columns that may be unsafe. For example, changing VARCHAR(20) to VARCHAR(30) is permitted, but changing it to VARCHAR(1024) is not because that alters the number of length bytes required to store individual values.

- Changing a `NULL` column to `NOT NULL` in non-strict mode is prohibited to prevent converting `NULL` values to default non-`NULL` values, for which there are no corresponding values in the referenced table. The operation is permitted in strict mode, but an error is returned if any such conversion is required.

`ALTER TABLE tbl_name RENAME new_tbl_name` changes internally generated foreign key constraint names and user-defined foreign key constraint names that begin with the string "*`tbl_name`*_ibfk_" to reflect the new table name. `InnoDB` interprets foreign key constraint names that begin with the string "*`tbl_name`*_ibfk_" as internally generated names.

Prior to MySQL 8.0.16, `ALTER TABLE` permits only the following limited version of `CHECK` constraint-adding syntax, which is parsed and ignored:

```
ADD CHECK (expr)
```

As of MySQL 8.0.16, `ALTER TABLE` permits `CHECK` constraints for existing tables to be added, dropped, or altered:

- Add a new `CHECK` constraint:

  ```
  ALTER TABLE tbl_name
      ADD [CONSTRAINT [symbol]] CHECK (expr) [[NOT] ENFORCED];
  ```

  The meaning of constraint syntax elements is the same as for `CREATE TABLE`. See Section 13.1.20.6, "CHECK Constraints".

- Drop an existing `CHECK` constraint named *`symbol`*:

  ```
  ALTER TABLE tbl_name
      DROP CHECK symbol;
  ```

- Alter whether an existing `CHECK` constraint named *`symbol`* is enforced:

  ```
  ALTER TABLE tbl_name
      ALTER CHECK symbol [NOT] ENFORCED;
  ```

The `DROP CHECK` and `ALTER CHECK` clauses are MySQL extensions to standard SQL.

As of MySQL 8.0.19, `ALTER TABLE` permits more general (and SQL standard) syntax for dropping and altering existing constraints of any type, where the constraint type is determined from the constraint name:

- Drop an existing constraint named *`symbol`*:

  ```
  ALTER TABLE tbl_name
      DROP CONSTRAINT symbol;
  ```

  If the `sql_require_primary_key` system variable is enabled, attempting to drop a primary key produces an error.

- Alter whether an existing constraint named *`symbol`* is enforced:

  ```
  ALTER TABLE tbl_name
      ALTER CONSTRAINT symbol [NOT] ENFORCED;
  ```

  Only `CHECK` constraints can be altered to be unenforced. All other constraint types are always enforced.

The SQL standard specifies that all types of constraints (primary key, unique index, foreign key, check) belong to the same namespace. In MySQL, each constraint type has its own namespace per schema. Consequently, names for each type of constraint must be unique per schema, but constraints of different types can have the same name. When multiple constraints have the same name, `DROP CONSTRAINT` and `ADD CONSTRAINT` are ambiguous and an error occurs. In such cases, constraint-specific syntax must be used to modify the constraint. For example, use `DROP PRIMARY KEY` or DROP FOREIGN KEY to drop a primary key or foreign key.

If a table alteration causes a violation of an enforced CHECK constraint, an error occurs and the table is not modified. Examples of operations for which an error occurs:

- Attempts to add the AUTO_INCREMENT attribute to a column that is used in a CHECK constraint.
- Attempts to add an enforced CHECK constraint or enforce a nonenforced CHECK constraint for which existing rows violate the constraint condition.
- Attempts to modify, rename, or drop a column that is used in a CHECK constraint, unless that constraint is also dropped in the same statement. Exception: If a CHECK constraint refers only to a single column, dropping the column automatically drops the constraint.

ALTER TABLE *tbl_name* RENAME *new_tbl_name* changes internally generated and user-defined CHECK constraint names that begin with the string "*tbl_name*_chk_" to reflect the new table name. MySQL interprets CHECK constraint names that begin with the string "*tbl_name*_chk_" as internally generated names.

## Changing the Character Set

To change the table default character set and all character columns (CHAR, VARCHAR, TEXT) to a new character set, use a statement like this:

```
ALTER TABLE tbl_name CONVERT TO CHARACTER SET charset_name;
```

The statement also changes the collation of all character columns. If you specify no COLLATE clause to indicate which collation to use, the statement uses default collation for the character set. If this collation is inappropriate for the intended table use (for example, if it would change from a case-sensitive collation to a case-insensitive collation), specify a collation explicitly.

For a column that has a data type of VARCHAR or one of the TEXT types, CONVERT TO CHARACTER SET changes the data type as necessary to ensure that the new column is long enough to store as many characters as the original column. For example, a TEXT column has two length bytes, which store the byte-length of values in the column, up to a maximum of 65,535. For a latin1 TEXT column, each character requires a single byte, so the column can store up to 65,535 characters. If the column is converted to utf8mb4, each character might require up to 4 bytes, for a maximum possible length of 4 × 65,535 = 262,140 bytes. That length does not fit in a TEXT column's length bytes, so MySQL converts the data type to MEDIUMTEXT, which is the smallest string type for which the length bytes can record a value of 262,140. Similarly, a VARCHAR column might be converted to MEDIUMTEXT.

To avoid data type changes of the type just described, do not use CONVERT TO CHARACTER SET. Instead, use MODIFY to change individual columns. For example:

```
ALTER TABLE t MODIFY latin1_text_col TEXT CHARACTER SET utf8mb4;
ALTER TABLE t MODIFY latin1_varchar_col VARCHAR(M) CHARACTER SET utf8mb4;
```

If you specify CONVERT TO CHARACTER SET binary, the CHAR, VARCHAR, and TEXT columns are converted to their corresponding binary string types (BINARY, VARBINARY, BLOB). This means that the columns no longer have a character set and a subsequent CONVERT TO operation does not apply to them.

If *charset_name* is DEFAULT in a CONVERT TO CHARACTER SET operation, the character set named by the character_set_database system variable is used.

**Warning**

The CONVERT TO operation converts column values between the original and named character sets. This is *not* what you want if you have a column in one character set (like latin1) but the stored values actually use some other, incompatible character set (like utf8mb4). In this case, you have to do the following for each such column:

```
ALTER TABLE t1 CHANGE c1 c1 BLOB;
ALTER TABLE t1 CHANGE c1 c1 TEXT CHARACTER SET utf8mb4;
```

> The reason this works is that there is no conversion when you convert to or from `BLOB` columns.

To change only the *default* character set for a table, use this statement:

```
ALTER TABLE tbl_name DEFAULT CHARACTER SET charset_name;
```

The word `DEFAULT` is optional. The default character set is the character set that is used if you do not specify the character set for columns that you add to a table later (for example, with `ALTER TABLE ... ADD column`).

When the `foreign_key_checks` system variable is enabled, which is the default setting, character set conversion is not permitted on tables that include a character string column used in a foreign key constraint. The workaround is to disable `foreign_key_checks` before performing the character set conversion. You must perform the conversion on both tables involved in the foreign key constraint before re-enabling `foreign_key_checks`. If you re-enable `foreign_key_checks` after converting only one of the tables, an `ON DELETE CASCADE` or `ON UPDATE CASCADE` operation could corrupt data in the referencing table due to implicit conversion that occurs during these operations (Bug #45290, Bug #74816).

## Importing InnoDB Tables

An `InnoDB` table created in its own file-per-table tablespace can be imported from a backup or from another MySQL server instance using `DISCARD TABLEPACE` and `IMPORT TABLESPACE` clauses. See Section 15.6.1.3, “Importing InnoDB Tables”.

## Row Order for MyISAM Tables

`ORDER BY` enables you to create the new table with the rows in a specific order. This option is useful primarily when you know that you query the rows in a certain order most of the time. By using this option after major changes to the table, you might be able to get higher performance. In some cases, it might make sorting easier for MySQL if the table is in order by the column that you want to order it by later.

> **Note**
>
> The table does not remain in the specified order after inserts and deletes.

`ORDER BY` syntax permits one or more column names to be specified for sorting, each of which optionally can be followed by `ASC` or `DESC` to indicate ascending or descending sort order, respectively. The default is ascending order. Only column names are permitted as sort criteria; arbitrary expressions are not permitted. This clause should be given last after any other clauses.

`ORDER BY` does not make sense for `InnoDB` tables because `InnoDB` always orders table rows according to the clustered index.

When used on a partitioned table, `ALTER TABLE ... ORDER BY` orders rows within each partition only.

## Partitioning Options

*`partition_options`* signifies options that can be used with partitioned tables for repartitioning, to add, drop, discard, import, merge, and split partitions, and to perform partitioning maintenance.

It is possible for an `ALTER TABLE` statement to contain a `PARTITION BY` or `REMOVE PARTITIONING` clause in an addition to other alter specifications, but the `PARTITION BY` or `REMOVE PARTITIONING` clause must be specified last after any other specifications. The `ADD PARTITION`, `DROP PARTITION`, `DISCARD PARTITION`, `IMPORT PARTITION`, `COALESCE PARTITION`, `REORGANIZE PARTITION`, `EXCHANGE PARTITION`, `ANALYZE PARTITION`, `CHECK PARTITION`, and `REPAIR PARTITION` options cannot be combined with other alter specifications in a single `ALTER TABLE`, since the options just listed act on individual partitions.

For more information about partition options, see Section 13.1.20, “CREATE TABLE Statement”, and Section 13.1.9.1, “ALTER TABLE Partition Operations”. For information about and examples of `ALTER TABLE ... EXCHANGE PARTITION` statements, see Section 24.3.3, “Exchanging Partitions and Subpartitions with Tables”.

### 13.1.9.1 ALTER TABLE Partition Operations

Partitioning-related clauses for `ALTER TABLE` can be used with partitioned tables for repartitioning, to add, drop, discard, import, merge, and split partitions, and to perform partitioning maintenance.

- Simply using a *`partition_options`* clause with `ALTER TABLE` on a partitioned table repartitions the table according to the partitioning scheme defined by the *`partition_options`*. This clause always begins with `PARTITION BY`, and follows the same syntax and other rules as apply to the *`partition_options`* clause for `CREATE TABLE` (for more detailed information, see Section 13.1.20, “CREATE TABLE Statement”), and can also be used to partition an existing table that is not already partitioned. For example, consider a (nonpartitioned) table defined as shown here:

```
CREATE TABLE t1 (
    id INT,
    year_col INT
);
```

This table can be partitioned by `HASH`, using the `id` column as the partitioning key, into 8 partitions by means of this statement:

```
ALTER TABLE t1
    PARTITION BY HASH(id)
    PARTITIONS 8;
```

MySQL supports an `ALGORITHM` option with `[SUB]PARTITION BY [LINEAR] KEY`. `ALGORITHM=1` causes the server to use the same key-hashing functions as MySQL 5.1 when computing the placement of rows in partitions; `ALGORITHM=2` means that the server employs the key-hashing functions implemented and used by default for new `KEY` partitioned tables in MySQL 5.5 and later. (Partitioned tables created with the key-hashing functions employed in MySQL 5.5 and later cannot be used by a MySQL 5.1 server.) Not specifying the option has the same effect as using `ALGORITHM=2`. This option is intended for use chiefly when upgrading or downgrading `[LINEAR] KEY` partitioned tables between MySQL 5.1 and later MySQL versions, or for creating tables partitioned by `KEY` or `LINEAR KEY` on a MySQL 5.5 or later server which can be used on a MySQL 5.1 server.

The table that results from using an `ALTER TABLE ... PARTITION BY` statement must follow the same rules as one created using `CREATE TABLE ... PARTITION BY`. This includes the rules governing the relationship between any unique keys (including any primary key) that the table might have, and the column or columns used in the partitioning expression, as discussed in Section 24.6.1, “Partitioning Keys, Primary Keys, and Unique Keys”. The `CREATE TABLE ... PARTITION BY` rules for specifying the number of partitions also apply to `ALTER TABLE ... PARTITION BY`.

The *`partition_definition`* clause for `ALTER TABLE ADD PARTITION` supports the same options as the clause of the same name for the `CREATE TABLE` statement. (See Section 13.1.20, “CREATE TABLE Statement”, for the syntax and description.) Suppose that you have the partitioned table created as shown here:

```
CREATE TABLE t1 (
    id INT,
    year_col INT
)
PARTITION BY RANGE (year_col) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1995),
    PARTITION p2 VALUES LESS THAN (1999)
);
```

You can add a new partition `p3` to this table for storing values less than `2002` as follows:

```
ALTER TABLE t1 ADD PARTITION (PARTITION p3 VALUES LESS THAN (2002));
```

`DROP PARTITION` can be used to drop one or more `RANGE` or `LIST` partitions. This statement cannot be used with `HASH` or `KEY` partitions; instead, use `COALESCE PARTITION` (see later in this section). Any data that was stored in the dropped partitions named in the *`partition_names`* list is discarded. For example, given the table `t1` defined previously, you can drop the partitions named `p0` and `p1` as shown here:

```
ALTER TABLE t1 DROP PARTITION p0, p1;
```

**Note**

`DROP PARTITION` does not work with tables that use the `NDB` storage engine. See Section 24.3.1, “Management of RANGE and LIST Partitions”, and Section 23.2.7, “Known Limitations of NDB Cluster”.

`ADD PARTITION` and `DROP PARTITION` do not currently support `IF [NOT] EXISTS`.

The `DISCARD PARTITION ... TABLESPACE` and `IMPORT PARTITION ... TABLESPACE` options extend the Transportable Tablespace feature to individual `InnoDB` table partitions. Each `InnoDB` table partition has its own tablespace file (`.ibd` file). The Transportable Tablespace feature makes it easy to copy the tablespaces from a running MySQL server instance to another running instance, or to perform a restore on the same instance. Both options take a comma-separated list of one or more partition names. For example:

```
ALTER TABLE t1 DISCARD PARTITION p2, p3 TABLESPACE;
```

```
ALTER TABLE t1 IMPORT PARTITION p2, p3 TABLESPACE;
```

When running `DISCARD PARTITION ... TABLESPACE` and `IMPORT PARTITION ... TABLESPACE` on subpartitioned tables, both partition and subpartition names are allowed. When a partition name is specified, subpartitions of that partition are included.

The Transportable Tablespace feature also supports copying or restoring partitioned `InnoDB` tables. For more information, see Section 15.6.1.3, “Importing InnoDB Tables”.

Renames of partitioned tables are supported. You can rename individual partitions indirectly using `ALTER TABLE ... REORGANIZE PARTITION`; however, this operation copies the partition's data.

To delete rows from selected partitions, use the `TRUNCATE PARTITION` option. This option takes a list of one or more comma-separated partition names. Consider the table `t1` created by this statement:

```
CREATE TABLE t1 (
    id INT,
    year_col INT
)
PARTITION BY RANGE (year_col) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1995),
    PARTITION p2 VALUES LESS THAN (1999),
    PARTITION p3 VALUES LESS THAN (2003),
    PARTITION p4 VALUES LESS THAN (2007)
```

```
);
```

To delete all rows from partition `p0`, use the following statement:

```
ALTER TABLE t1 TRUNCATE PARTITION p0;
```

The statement just shown has the same effect as the following `DELETE` statement:

```
DELETE FROM t1 WHERE year_col < 1991;
```

When truncating multiple partitions, the partitions do not have to be contiguous: This can greatly simplify delete operations on partitioned tables that would otherwise require very complex `WHERE` conditions if done with `DELETE` statements. For example, this statement deletes all rows from partitions `p1` and `p3`:

```
ALTER TABLE t1 TRUNCATE PARTITION p1, p3;
```

An equivalent `DELETE` statement is shown here:

```
DELETE FROM t1 WHERE
    (year_col >= 1991 AND year_col < 1995)
    OR
    (year_col >= 2003 AND year_col < 2007);
```

If you use the `ALL` keyword in place of the list of partition names, the statement acts on all table partitions.

`TRUNCATE PARTITION` merely deletes rows; it does not alter the definition of the table itself, or of any of its partitions.

To verify that the rows were dropped, check the `INFORMATION_SCHEMA.PARTITIONS` table, using a query such as this one:

```
SELECT PARTITION_NAME, TABLE_ROWS
    FROM INFORMATION_SCHEMA.PARTITIONS
    WHERE TABLE_NAME = 't1';
```

`COALESCE PARTITION` can be used with a table that is partitioned by `HASH` or `KEY` to reduce the number of partitions by *`number`*. Suppose that you have created table `t2` as follows:

```
CREATE TABLE t2 (
    name VARCHAR (30),
    started DATE
)
PARTITION BY HASH( YEAR(started) )
PARTITIONS 6;
```

To reduce the number of partitions used by `t2` from 6 to 4, use the following statement:

```
ALTER TABLE t2 COALESCE PARTITION 2;
```

The data contained in the last *`number`* partitions is merged into the remaining partitions. In this case, partitions 4 and 5 are merged into the first 4 partitions (the partitions numbered 0, 1, 2, and 3).

To change some but not all the partitions used by a partitioned table, you can use `REORGANIZE PARTITION`. This statement can be used in several ways:

- To merge a set of partitions into a single partition. This is done by naming several partitions in the *`partition_names`* list and supplying a single definition for *`partition_definition`*.
- To split an existing partition into several partitions. Accomplish this by naming a single partition for *`partition_names`* and providing multiple *`partition_definitions`*.

- To change the ranges for a subset of partitions defined using `VALUES LESS THAN` or the value lists for a subset of partitions defined using `VALUES IN`.

**Note**

For partitions that have not been explicitly named, MySQL automatically provides the default names `p0`, `p1`, `p2`, and so on. The same is true with regard to subpartitions.

For more detailed information about and examples of `ALTER TABLE ... REORGANIZE PARTITION` statements, see Section 24.3.1, “Management of RANGE and LIST Partitions”.

- To exchange a table partition or subpartition with a table, use the `ALTER TABLE ... EXCHANGE PARTITION` statement—that is, to move any existing rows in the partition or subpartition to the nonpartitioned table, and any existing rows in the nonpartitioned table to the table partition or subpartition.

  Once one or more columns have been added to a partitioned table using `ALGORITHM=INSTANT`, it is no longer possible to exchange partitions with that table.

  For usage information and examples, see Section 24.3.3, “Exchanging Partitions and Subpartitions with Tables”.

- Several options provide partition maintenance and repair functionality analogous to that implemented for nonpartitioned tables by statements such as `CHECK TABLE` and `REPAIR TABLE` (which are also supported for partitioned tables; for more information, see Section 13.7.3, “Table Maintenance Statements”). These include `ANALYZE PARTITION`, `CHECK PARTITION`, `OPTIMIZE PARTITION`, `REBUILD PARTITION`, and `REPAIR PARTITION`. Each of these options takes a *`partition_names`* clause consisting of one or more names of partitions, separated by commas. The partitions must already exist in the target table. You can also use the `ALL` keyword in place of *`partition_names`*, in which case the statement acts on all table partitions. For more information and examples, see Section 24.3.4, “Maintenance of Partitions”.

  `InnoDB` does not currently support per-partition optimization; `ALTER TABLE ... OPTIMIZE PARTITION` causes the entire table to rebuilt and analyzed, and an appropriate warning to be issued. (Bug #11751825, Bug #42822) To work around this problem, use `ALTER TABLE ... REBUILD PARTITION` and `ALTER TABLE ... ANALYZE PARTITION` instead.

  The `ANALYZE PARTITION`, `CHECK PARTITION`, `OPTIMIZE PARTITION`, and `REPAIR PARTITION` options are not supported for tables which are not partitioned.

- `REMOVE PARTITIONING` enables you to remove a table's partitioning without otherwise affecting the table or its data. This option can be combined with other `ALTER TABLE` options such as those used to add, drop, or rename columns or indexes.

- Using the `ENGINE` option with `ALTER TABLE` changes the storage engine used by the table without affecting the partitioning. The target storage engine must provide its own partitioning handler. Only the `InnoDB` and `NDB` storage engines have native partitioning handlers; `NDB` is not currently supported in MySQL 8.0.

It is possible for an `ALTER TABLE` statement to contain a `PARTITION BY` or `REMOVE PARTITIONING` clause in an addition to other alter specifications, but the `PARTITION BY` or `REMOVE PARTITIONING` clause must be specified last after any other specifications.

The `ADD PARTITION`, `DROP PARTITION`, `COALESCE PARTITION`, `REORGANIZE PARTITION`, `ANALYZE PARTITION`, `CHECK PARTITION`, and `REPAIR PARTITION` options cannot be combined with other alter specifications in a single `ALTER TABLE`, since the options just listed act on individual partitions. For more information, see Section 13.1.9.1, “ALTER TABLE Partition Operations”.

Only a single instance of any one of the following options can be used in a given `ALTER TABLE` statement: `PARTITION BY`, `ADD PARTITION`, `DROP PARTITION`, `TRUNCATE PARTITION`, `EXCHANGE PARTITION`, `REORGANIZE PARTITION`, or `COALESCE PARTITION`, `ANALYZE PARTITION`, `CHECK PARTITION`, `OPTIMIZE PARTITION`, `REBUILD PARTITION`, `REMOVE PARTITIONING`.

For example, the following two statements are invalid:

```
ALTER TABLE t1 ANALYZE PARTITION p1, ANALYZE PARTITION p2;

ALTER TABLE t1 ANALYZE PARTITION p1, CHECK PARTITION p2;
```

In the first case, you can analyze partitions `p1` and `p2` of table `t1` concurrently using a single statement with a single `ANALYZE PARTITION` option that lists both of the partitions to be analyzed, like this:

```
ALTER TABLE t1 ANALYZE PARTITION p1, p2;
```

In the second case, it is not possible to perform `ANALYZE` and `CHECK` operations on different partitions of the same table concurrently. Instead, you must issue two separate statements, like this:

```
ALTER TABLE t1 ANALYZE PARTITION p1;
ALTER TABLE t1 CHECK PARTITION p2;
```

`REBUILD` operations are currently unsupported for subpartitions. The `REBUILD` keyword is expressly disallowed with subpartitions, and causes `ALTER TABLE` to fail with an error if so used.

`CHECK PARTITION` and `REPAIR PARTITION` operations fail when the partition to be checked or repaired contains any duplicate key errors.

For more information about these statements, see Section 24.3.4, “Maintenance of Partitions”.

### 13.1.9.2 ALTER TABLE and Generated Columns

`ALTER TABLE` operations permitted for generated columns are `ADD`, `MODIFY`, and `CHANGE`.

- Generated columns can be added.

  ```
  CREATE TABLE t1 (c1 INT);
  ALTER TABLE t1 ADD COLUMN c2 INT GENERATED ALWAYS AS (c1 + 1) STORED;
  ```

- The data type and expression of generated columns can be modified.

  ```
  CREATE TABLE t1 (c1 INT, c2 INT GENERATED ALWAYS AS (c1 + 1) STORED);
  ALTER TABLE t1 MODIFY COLUMN c2 TINYINT GENERATED ALWAYS AS (c1 + 5) STORED;
  ```

- Generated columns can be renamed or dropped, if no other column refers to them.

  ```
  CREATE TABLE t1 (c1 INT, c2 INT GENERATED ALWAYS AS (c1 + 1) STORED);
  ALTER TABLE t1 CHANGE c2 c3 INT GENERATED ALWAYS AS (c1 + 1) STORED;
  ALTER TABLE t1 DROP COLUMN c3;
  ```

- Virtual generated columns cannot be altered to stored generated columns, or vice versa. To work around this, drop the column, then add it with the new definition.

  ```
  CREATE TABLE t1 (c1 INT, c2 INT GENERATED ALWAYS AS (c1 + 1) VIRTUAL);
  ALTER TABLE t1 DROP COLUMN c2;
  ALTER TABLE t1 ADD COLUMN c2 INT GENERATED ALWAYS AS (c1 + 1) STORED;
  ```

- Nongenerated columns can be altered to stored but not virtual generated columns.

  ```
  CREATE TABLE t1 (c1 INT, c2 INT);
  ALTER TABLE t1 MODIFY COLUMN c2 INT GENERATED ALWAYS AS (c1 + 1) STORED;
  ```

- Stored but not virtual generated columns can be altered to nongenerated columns. The stored generated values become the values of the nongenerated column.

  ```
  CREATE TABLE t1 (c1 INT, c2 INT GENERATED ALWAYS AS (c1 + 1) STORED);
  ALTER TABLE t1 MODIFY COLUMN c2 INT;
  ```

- `ADD COLUMN` is not an in-place operation for stored columns (done without using a temporary table) because the expression must be evaluated by the server. For stored columns, indexing changes are done in place, and expression changes are not done in place. Changes to column comments are done in place.
- For non-partitioned tables, `ADD COLUMN` and `DROP COLUMN` are in-place operations for virtual columns. However, adding or dropping a virtual column cannot be performed in place in combination with other `ALTER TABLE` operations.

  For partitioned tables, `ADD COLUMN` and `DROP COLUMN` are not in-place operations for virtual columns.
- `InnoDB` supports secondary indexes on virtual generated columns. Adding or dropping a secondary index on a virtual generated column is an in-place operation. For more information, see Section 13.1.20.9, “Secondary Indexes and Generated Columns”.
- When a `VIRTUAL` generated column is added to a table or modified, it is not ensured that data being calculated by the generated column expression is not out of range for the column. This can lead to inconsistent data being returned and unexpectedly failed statements. To permit control over whether validation occurs for such columns, `ALTER TABLE` supports `WITHOUT VALIDATION` and `WITH VALIDATION` clauses:
  - With `WITHOUT VALIDATION` (the default if neither clause is specified), an in-place operation is performed (if possible), data integrity is not checked, and the statement finishes more quickly. However, later reads from the table might report warnings or errors for the column if values are out of range.
  - With `WITH VALIDATION`, `ALTER TABLE` copies the table. If an out-of-range or any other error occurs, the statement fails. Because a table copy is performed, the statement takes longer.

  `WITHOUT VALIDATION` and `WITH VALIDATION` are permitted only with `ADD COLUMN`, `CHANGE COLUMN`, and `MODIFY COLUMN` operations. Otherwise, an `ER_WRONG_USAGE` error occurs.
- If expression evaluation causes truncation or provides incorrect input to a function, the `ALTER TABLE` statement terminates with an error and the DDL operation is rejected.
- An `ALTER TABLE` statement that changes the default value of a column *`col_name`* may also change the value of a generated column expression that refers to the column using *`col_name`*, which may change the value of a generated column expression that refers to the column using `DEFAULT(`*`col_name`*`)`. For this reason, `ALTER TABLE` operations that change the definition of a column cause a table rebuild if any generated column expression uses `DEFAULT()`.

### 13.1.9.3 ALTER TABLE Examples

Begin with a table `t1` created as shown here:

```
CREATE TABLE t1 (a INTEGER, b CHAR(10));
```

To rename the table from `t1` to `t2`:

```
ALTER TABLE t1 RENAME t2;
```

To change column `a` from `INTEGER` to `TINYINT NOT NULL` (leaving the name the same), and to change column `b` from `CHAR(10)` to `CHAR(20)` as well as renaming it from `b` to `c`:

```
ALTER TABLE t2 MODIFY a TINYINT NOT NULL, CHANGE b c CHAR(20);
```

To add a new `TIMESTAMP` column named `d`:

```
ALTER TABLE t2 ADD d TIMESTAMP;
```

To add an index on column `d` and a `UNIQUE` index on column `a`:

```
ALTER TABLE t2 ADD INDEX (d), ADD UNIQUE (a);
```

To remove column `c`:

```
ALTER TABLE t2 DROP COLUMN c;
```

To add a new `AUTO_INCREMENT` integer column named `c`:

```
ALTER TABLE t2 ADD c INT UNSIGNED NOT NULL AUTO_INCREMENT,
  ADD PRIMARY KEY (c);
```

We indexed `c` (as a `PRIMARY KEY`) because `AUTO_INCREMENT` columns must be indexed, and we declare `c` as `NOT NULL` because primary key columns cannot be `NULL`.

For `NDB` tables, it is also possible to change the storage type used for a table or column. For example, consider an `NDB` table created as shown here:

```
mysql> CREATE TABLE t1 (c1 INT) TABLESPACE ts_1 ENGINE NDB;
Query OK, 0 rows affected (1.27 sec)
```

To convert this table to disk-based storage, you can use the following `ALTER TABLE` statement:

```
mysql> ALTER TABLE t1 TABLESPACE ts_1 STORAGE DISK;
Query OK, 0 rows affected (2.99 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SHOW CREATE TABLE t1\G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `c1` int(11) DEFAULT NULL
) /*!50100 TABLESPACE ts_1 STORAGE DISK */
ENGINE=ndbcluster DEFAULT CHARSET=latin1
1 row in set (0.01 sec)
```

It is not necessary that the tablespace was referenced when the table was originally created; however, the tablespace must be referenced by the `ALTER TABLE`:

```
mysql> CREATE TABLE t2 (c1 INT) ts_1 ENGINE NDB;
Query OK, 0 rows affected (1.00 sec)

mysql> ALTER TABLE t2 STORAGE DISK;
ERROR 1005 (HY000): Can't create table 'c.#sql-1750_3' (errno: 140)
mysql> ALTER TABLE t2 TABLESPACE ts_1 STORAGE DISK;
Query OK, 0 rows affected (3.42 sec)
Records: 0  Duplicates: 0  Warnings: 0
mysql> SHOW CREATE TABLE t2\G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t2` (
  `c1` int(11) DEFAULT NULL
) /*!50100 TABLESPACE ts_1 STORAGE DISK */
ENGINE=ndbcluster DEFAULT CHARSET=latin1
1 row in set (0.01 sec)
```

To change the storage type of an individual column, you can use `ALTER TABLE ... MODIFY [COLUMN]`. For example, suppose you create an NDB Cluster Disk Data table with two columns, using this `CREATE TABLE` statement:

```
mysql> CREATE TABLE t3 (c1 INT, c2 INT)
    ->     TABLESPACE ts_1 STORAGE DISK ENGINE NDB;
Query OK, 0 rows affected (1.34 sec)
```

To change column `c2` from disk-based to in-memory storage, include a STORAGE MEMORY clause in the column definition used by the ALTER TABLE statement, as shown here:

```
mysql> ALTER TABLE t3 MODIFY c2 INT STORAGE MEMORY;
Query OK, 0 rows affected (3.14 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

You can make an in-memory column into a disk-based column by using `STORAGE DISK` in a similar fashion.

Column `c1` uses disk-based storage, since this is the default for the table (determined by the table-level `STORAGE DISK` clause in the `CREATE TABLE` statement). However, column `c2` uses in-memory storage, as can be seen here in the output of SHOW `CREATE TABLE`:

```
mysql> SHOW CREATE TABLE t3\G
*************************** 1. row ***************************
       Table: t3
Create Table: CREATE TABLE `t3` (
  `c1` int(11) DEFAULT NULL,
  `c2` int(11) /*!50120 STORAGE MEMORY */ DEFAULT NULL
) /*!50100 TABLESPACE ts_1 STORAGE DISK */ ENGINE=ndbcluster DEFAULT CHARSET=latin1
1 row in set (0.02 sec)
```

When you add an `AUTO_INCREMENT` column, column values are filled in with sequence numbers automatically. For `MyISAM` tables, you can set the first sequence number by executing `SET INSERT_ID=value` before `ALTER TABLE` or by using the `AUTO_INCREMENT=value` table option.

With `MyISAM` tables, if you do not change the `AUTO_INCREMENT` column, the sequence number is not affected. If you drop an `AUTO_INCREMENT` column and then add another `AUTO_INCREMENT` column, the numbers are resequenced beginning with 1.

When replication is used, adding an `AUTO_INCREMENT` column to a table might not produce the same ordering of the rows on the replica and the source. This occurs because the order in which the rows are numbered depends on the specific storage engine used for the table and the order in which the rows were inserted. If it is important to have the same order on the source and replica, the rows must be ordered before assigning an `AUTO_INCREMENT` number. Assuming that you want to add an `AUTO_INCREMENT` column to the table `t1`, the following statements produce a new table `t2` identical to `t1` but with an `AUTO_INCREMENT` column:

```
CREATE TABLE t2 (id INT AUTO_INCREMENT PRIMARY KEY)
SELECT * FROM t1 ORDER BY col1, col2;
```

This assumes that the table `t1` has columns `col1` and `col2`.

This set of statements also produces a new table `t2` identical to `t1`, with the addition of an `AUTO_INCREMENT` column:

```
CREATE TABLE t2 LIKE t1;
ALTER TABLE t2 ADD id INT AUTO_INCREMENT PRIMARY KEY;
INSERT INTO t2 SELECT * FROM t1 ORDER BY col1, col2;
```

**Important**

To guarantee the same ordering on both source and replica, *all* columns of `t1` must be referenced in the `ORDER BY` clause.

Regardless of the method used to create and populate the copy having the `AUTO_INCREMENT` column, the final step is to drop the original table and then rename the copy:

```
DROP TABLE t1;
ALTER TABLE t2 RENAME t1;
```

## 13.1.10 ALTER TABLESPACE Statement

```
ALTER [UNDO] TABLESPACE tablespace_name
  NDB only:
    {ADD | DROP} DATAFILE 'file_name'
    [INITIAL_SIZE [=] size]
    [WAIT]
  InnoDB and NDB:
    [RENAME TO tablespace_name]
  InnoDB only:
    [AUTOEXTEND_SIZE [=] 'value']
    [SET {ACTIVE | INACTIVE}]
    [ENCRYPTION [=] {'Y' | 'N'}]
  InnoDB and NDB:
    [ENGINE [=] engine_name]
```

```
  Reserved for future use:
    [ENGINE_ATTRIBUTE [=] 'string']
```

This statement is used with NDB and InnoDB tablespaces. It can be used to add a new data file to, or to drop a data file from an NDB tablespace. It can also be used to rename an NDB Cluster Disk Data tablespace, rename an InnoDB general tablespace, encrypt an InnoDB general tablespace, or mark an InnoDB undo tablespace as active or inactive.

The UNDO keyword, introduced in MySQL 8.0.14, is used with the SET {ACTIVE | INACTIVE} clause to mark an InnoDB undo tablespace as active or inactive. For more information, see Section 15.6.3.4, “Undo Tablespaces”.

The ADD DATAFILE variant enables you to specify an initial size for an NDB Disk Data tablespace using an INITIAL_SIZE clause, where *size* is measured in bytes; the default value is 134217728 (128 MB). You may optionally follow *size* with a one-letter abbreviation for an order of magnitude, similar to those used in my.cnf. Generally, this is one of the letters M (megabytes) or G (gigabytes).

On 32-bit systems, the maximum supported value for INITIAL_SIZE is 4294967296 (4 GB). (Bug #29186)

INITIAL_SIZE is rounded, explicitly, as for CREATE TABLESPACE.

Once a data file has been created, its size cannot be changed; however, you can add more data files to an NDB tablespace using additional ALTER TABLESPACE ... ADD DATAFILE statements.

When ALTER TABLESPACE ... ADD DATAFILE is used with ENGINE = NDB, a data file is created on each Cluster data node, but only one row is generated in the Information Schema FILES table. See the description of this table, as well as Section 23.6.11.1, “NDB Cluster Disk Data Objects”, for more information. ADD DATAFILE is not supported with InnoDB tablespaces.

Using DROP DATAFILE with ALTER TABLESPACE drops the data file '*file_name*' from an NDB tablespace. You cannot drop a data file from a tablespace which is in use by any table; in other words, the data file must be empty (no extents used). See Section 23.6.11.1, “NDB Cluster Disk Data Objects”. In addition, any data file to be dropped must previously have been added to the tablespace with CREATE TABLESPACE or ALTER TABLESPACE. DROP DATAFILE is not supported with InnoDB tablespaces.

WAIT is parsed but otherwise ignored. It is intended for future expansion.

The ENGINE clause, which specifies the storage engine used by the tablespace, is deprecated; expect it to be removed in a future release. The tablespace storage engine is known by the data dictionary, making the ENGINE clause obsolete. If the storage engine is specified, it must match the tablespace storage engine defined in the data dictionary. The only values for *engine_name* compatible with NDB tablespaces are NDB and NDBCLUSTER.

RENAME TO operations are implicitly performed in autocommit mode, regardless of the autocommit setting.

A RENAME TO operation cannot be performed while LOCK TABLES or FLUSH TABLES WITH READ LOCK is in effect for tables that reside in the tablespace.

Exclusive metadata locks are taken on tables that reside in a general tablespace while the tablespace is renamed, which prevents concurrent DDL. Concurrent DML is supported.

The CREATE TABLESPACE privilege is required to rename an InnoDB general tablespace.

The AUTOEXTEND_SIZE option defines the amount by which InnoDB extends the size of a tablespace when it becomes full. Introduced in MySQL 8.0.23. The setting must be a multiple of 4MB. The default setting is 0, which causes the tablespace to be extended according to the implicit default behavior. For more information, see Section 15.6.3.9, “Tablespace AUTOEXTEND_SIZE Configuration”.

The ENCRYPTION clause enables or disables page-level data encryption for an InnoDB general tablespace or the mysql system tablespace. Encryption support for general tablespaces was

introduced in MySQL 8.0.13. Encryption support for the `mysql` system tablespace was introduced in MySQL 8.0.16.

A keyring plugin must be installed and configured before encryption can be enabled.

As of MySQL 8.0.16, if the `table_encryption_privilege_check` variable is enabled, the `TABLE_ENCRYPTION_ADMIN` privilege is required to alter a general tablespace with an `ENCRYPTION` clause setting that differs from the `default_table_encryption` setting.

Enabling encryption for a general tablespace fails if any table in the tablespace belongs to a schema defined with `DEFAULT ENCRYPTION='N'`. Similarly, disabling encryption fails if any table in the general tablespace belongs to a schema defined with `DEFAULT ENCRYPTION='Y'`. The `DEFAULT ENCRYPTION` schema option was introduced in MySQL 8.0.16.

If an `ALTER TABLESPACE` statement executed on a general tablespace does not include an `ENCRYPTION` clause, the tablespace retains its current encryption status, regardless of the `default_table_encryption` setting.

When a general tablespace or the `mysql` system tablespace is encrypted, all tables residing in the tablespace are encrypted. Likewise, a table created in an encrypted tablespace is encrypted.

The `INPLACE` algorithm is used when altering the `ENCRYPTION` attribute of a general tablespace or the `mysql` system tablespace. The `INPLACE` algorithm permits concurrent DML on tables that reside in the tablespace. Concurrent DDL is blocked.

For more information, see Section 15.13, “InnoDB Data-at-Rest Encryption”.

The `ENGINE_ATTRIBUTE` option (available as of MySQL 8.0.21) is used to specify tablespace attributes for primary storage engines. The option is reserved for future use.

Permitted values are a string literal containing a valid `JSON` document or an empty string (''). Invalid `JSON` is rejected.

```
ALTER TABLESPACE ts1 ENGINE_ATTRIBUTE='{"key":"value"}';
```

`ENGINE_ATTRIBUTE` values can be repeated without error. In this case, the last specified value is used.

`ENGINE_ATTRIBUTE` values are not checked by the server, nor are they cleared when the table's storage engine is changed.

It is not permitted to alter an individual element of a JSON attribute value. You can only add or replace an attribute.

## 13.1.11 ALTER VIEW Statement

```
ALTER
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = user]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

This statement changes the definition of a view, which must exist. The syntax is similar to that for `CREATE VIEW` see Section 13.1.23, “CREATE VIEW Statement”). This statement requires the `CREATE VIEW` and `DROP` privileges for the view, and some privilege for each column referred to in the `SELECT` statement. `ALTER VIEW` is permitted only to the definer or users with the `SET_USER_ID` privilege (or the deprecated `SUPER` privilege).

## 13.1.12 CREATE DATABASE Statement

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
    [create_option] ...
```

```
create_option: [DEFAULT] {
    CHARACTER SET [=] charset_name
  | COLLATE [=] collation_name
  | ENCRYPTION [=] {'Y' | 'N'}
}
```

`CREATE DATABASE` creates a database with the given name. To use this statement, you need the `CREATE` privilege for the database. `CREATE SCHEMA` is a synonym for `CREATE DATABASE`.

An error occurs if the database exists and you did not specify `IF NOT EXISTS`.

`CREATE DATABASE` is not permitted within a session that has an active `LOCK TABLES` statement.

Each *`create_option`* specifies a database characteristic. Database characteristics are stored in the data dictionary.

- The `CHARACTER SET` option specifies the default database character set. The `COLLATE` option specifies the default database collation. For information about character set and collation names, see Chapter 10, *Character Sets, Collations, Unicode*.

  To see the available character sets and collations, use the the `SHOW CHARACTER SET` and `SHOW COLLATION` statements, respectively. See Section 13.7.7.3, “SHOW CHARACTER SET Statement”, and Section 13.7.7.4, “SHOW COLLATION Statement”.

- The `ENCRYPTION` option, introduced in MySQL 8.0.16, defines the default database encryption, which is inherited by tables created in the database. The permitted values are `'Y'` (encryption enabled) and `'N'` (encryption disabled). If the `ENCRYPTION` option is not specified, the value of the `default_table_encryption` system variable defines the default database encryption. If the `table_encryption_privilege_check` system variable is enabled, the `TABLE_ENCRYPTION_ADMIN` privilege is required to specify a default encryption setting that differs from the `default_table_encryption` setting. For more information, see Defining an Encryption Default for Schemas and General Tablespaces.

A database in MySQL is implemented as a directory containing files that correspond to tables in the database. Because there are no tables in a database when it is initially created, the `CREATE DATABASE` statement creates only a directory under the MySQL data directory. Rules for permissible database names are given in Section 9.2, “Schema Object Names”. If a database name contains special characters, the name for the database directory contains encoded versions of those characters as described in Section 9.2.4, “Mapping of Identifiers to File Names”.

Creating a database directory by manually creating a directory under the data directory (for example, with `mkdir`) is unsupported in MySQL 8.0.

When you create a database, let the server manage the directory and the files in it. Manipulating database directories and files directly can cause inconsistencies and unexpected results.

MySQL has no limit on the number of databases. The underlying file system may have a limit on the number of directories.

You can also use the `mysqladmin` program to create databases. See Section 4.5.2, “mysqladmin — A MySQL Server Administration Program”.

## 13.1.13 CREATE EVENT Statement

```
CREATE
    [DEFINER = user]
    EVENT
    [IF NOT EXISTS]
    event_name
    ON SCHEDULE schedule
    [ON COMPLETION [NOT] PRESERVE]
    [ENABLE | DISABLE | DISABLE ON SLAVE]
    [COMMENT 'string']
    DO event_body;
```

```
schedule: {
    AT timestamp [+ INTERVAL interval] ...
  | EVERY interval
    [STARTS timestamp [+ INTERVAL interval] ...]
    [ENDS timestamp [+ INTERVAL interval] ...]
}

interval:
    quantity {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |
              WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE |
              DAY_SECOND | HOUR_MINUTE | HOUR_SECOND | MINUTE_SECOND}
```

This statement creates and schedules a new event. The event does not run unless the Event Scheduler is enabled. For information about checking Event Scheduler status and enabling it if necessary, see Section 25.4.2, “Event Scheduler Configuration”.

`CREATE EVENT` requires the `EVENT` privilege for the schema in which the event is to be created. If the `DEFINER` clause is present, the privileges required depend on the *`user`* value, as discussed in Section 25.6, “Stored Object Access Control”.

The minimum requirements for a valid `CREATE EVENT` statement are as follows:

- The keywords `CREATE EVENT` plus an event name, which uniquely identifies the event in a database schema.
- An `ON SCHEDULE` clause, which determines when and how often the event executes.
- A `DO` clause, which contains the SQL statement to be executed by an event.

This is an example of a minimal `CREATE EVENT` statement:

```
CREATE EVENT myevent
    ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 1 HOUR
    DO
      UPDATE myschema.mytable SET mycol = mycol + 1;
```

The previous statement creates an event named `myevent`. This event executes once—one hour following its creation—by running an SQL statement that increments the value of the `myschema.mytable` table's `mycol` column by 1.

The *`event_name`* must be a valid MySQL identifier with a maximum length of 64 characters. Event names are not case-sensitive, so you cannot have two events named `myevent` and `MyEvent` in the same schema. In general, the rules governing event names are the same as those for names of stored routines. See Section 9.2, “Schema Object Names”.

An event is associated with a schema. If no schema is indicated as part of *`event_name`*, the default (current) schema is assumed. To create an event in a specific schema, qualify the event name with a schema using *`schema_name.event_name`* syntax.

The `DEFINER` clause specifies the MySQL account to be used when checking access privileges at event execution time. If the `DEFINER` clause is present, the *`user`* value should be a MySQL account specified as `'user_name'@'host_name'`, `CURRENT_USER`, or `CURRENT_USER()`. The permitted *`user`* values depend on the privileges you hold, as discussed in Section 25.6, “Stored Object Access Control”. Also see that section for additional information about event security.

If the `DEFINER` clause is omitted, the default definer is the user who executes the `CREATE EVENT` statement. This is the same as specifying `DEFINER = CURRENT_USER` explicitly.

Within an event body, the `CURRENT_USER` function returns the account used to check privileges at event execution time, which is the `DEFINER` user. For information about user auditing within events, see Section 6.2.23, “SQL-Based Account Activity Auditing”.

`IF NOT EXISTS` has the same meaning for `CREATE EVENT` as for `CREATE TABLE`: If an event named *`event_name`* already exists in the same schema, no action is taken, and no error results. (However, a warning is generated in such cases.)

The `ON SCHEDULE` clause determines when, how often, and for how long the *`event_body`* defined for the event repeats. This clause takes one of two forms:

- `AT` *`timestamp`* is used for a one-time event. It specifies that the event executes one time only at the date and time given by *`timestamp`*, which must include both the date and time, or must be an expression that resolves to a datetime value. You may use a value of either the `DATETIME` or `TIMESTAMP` type for this purpose. If the date is in the past, a warning occurs, as shown here:

```
mysql> SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2006-02-10 23:59:01 |
+---------------------+
1 row in set (0.04 sec)

mysql> CREATE EVENT e_totals
    ->     ON SCHEDULE AT '2006-02-10 23:59:00'
    ->     DO INSERT INTO test.totals VALUES (NOW());
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1588
Message: Event execution time is in the past and ON COMPLETION NOT
         PRESERVE is set. The event was dropped immediately after
         creation.
```

  `CREATE EVENT` statements which are themselves invalid—for whatever reason—fail with an error.

  You may use `CURRENT_TIMESTAMP` to specify the current date and time. In such a case, the event acts as soon as it is created.

  To create an event which occurs at some point in the future relative to the current date and time—such as that expressed by the phrase “three weeks from now”—you can use the optional clause `+ INTERVAL` *`interval`*. The *`interval`* portion consists of two parts, a quantity and a unit of time, and follows the syntax rules described in Temporal Intervals, except that you cannot use any units keywords that involving microseconds when defining an event. With some interval types, complex time units may be used. For example, “two minutes and ten seconds” can be expressed as `+ INTERVAL '2:10' MINUTE_SECOND`.

  You can also combine intervals. For example, `AT CURRENT_TIMESTAMP + INTERVAL 3 WEEK + INTERVAL 2 DAY` is equivalent to “three weeks and two days from now”. Each portion of such a clause must begin with `+ INTERVAL`.

- To repeat actions at a regular interval, use an `EVERY` clause. The `EVERY` keyword is followed by an *`interval`* as described in the previous discussion of the `AT` keyword. (`+ INTERVAL` is *not* used with `EVERY`.) For example, `EVERY 6 WEEK` means “every six weeks”.

  Although `+ INTERVAL` clauses are not permitted in an `EVERY` clause, you can use the same complex time units permitted in a `+ INTERVAL`.

  An `EVERY` clause may contain an optional `STARTS` clause. `STARTS` is followed by a *`timestamp`* value that indicates when the action should begin repeating, and may also use `+ INTERVAL` *`interval`* to specify an amount of time “from now”. For example, `EVERY 3 MONTH STARTS CURRENT_TIMESTAMP + INTERVAL 1 WEEK` means “every three months, beginning one week from now”. Similarly, you can express “every two weeks, beginning six hours and fifteen minutes from now” as `EVERY 2 WEEK STARTS CURRENT_TIMESTAMP + INTERVAL '6:15' HOUR_MINUTE`. Not specifying `STARTS` is the same as using `STARTS CURRENT_TIMESTAMP`—that is, the action specified for the event begins repeating immediately upon creation of the event.

  An `EVERY` clause may contain an optional `ENDS` clause. The `ENDS` keyword is followed by a *`timestamp`* value that tells MySQL when the event should stop repeating. You may also use `+`

`INTERVAL interval` with `ENDS`; for instance, `EVERY 12 HOUR STARTS CURRENT_TIMESTAMP + INTERVAL 30 MINUTE ENDS CURRENT_TIMESTAMP + INTERVAL 4 WEEK` is equivalent to "every twelve hours, beginning thirty minutes from now, and ending four weeks from now". Not using `ENDS` means that the event continues executing indefinitely.

`ENDS` supports the same syntax for complex time units as `STARTS` does.

You may use `STARTS`, `ENDS`, both, or neither in an `EVERY` clause.

If a repeating event does not terminate within its scheduling interval, the result may be multiple instances of the event executing simultaneously. If this is undesirable, you should institute a mechanism to prevent simultaneous instances. For example, you could use the `GET_LOCK()` function, or row or table locking.

The `ON SCHEDULE` clause may use expressions involving built-in MySQL functions and user variables to obtain any of the *`timestamp`* or *`interval`* values which it contains. You may not use stored functions or loadable functions in such expressions, nor may you use any table references; however, you may use `SELECT FROM DUAL`. This is true for both `CREATE EVENT` and `ALTER EVENT` statements. References to stored functions, loadable functions, and tables in such cases are specifically not permitted, and fail with an error (see Bug #22830).

Times in the `ON SCHEDULE` clause are interpreted using the current session `time_zone` value. This becomes the event time zone; that is, the time zone that is used for event scheduling and is in effect within the event as it executes. These times are converted to UTC and stored along with the event time zone internally. This enables event execution to proceed as defined regardless of any subsequent changes to the server time zone or daylight saving time effects. For additional information about representation of event times, see Section 25.4.4, "Event Metadata". See also Section 13.7.7.18, "SHOW EVENTS Statement", and Section 26.3.14, "The INFORMATION_SCHEMA EVENTS Table".

Normally, once an event has expired, it is immediately dropped. You can override this behavior by specifying `ON COMPLETION PRESERVE`. Using `ON COMPLETION NOT PRESERVE` merely makes the default nonpersistent behavior explicit.

You can create an event but prevent it from being active using the `DISABLE` keyword. Alternatively, you can use `ENABLE` to make explicit the default status, which is active. This is most useful in conjunction with `ALTER EVENT` (see Section 13.1.3, "ALTER EVENT Statement").

A third value may also appear in place of `ENABLE` or `DISABLE`; `DISABLE ON SLAVE` is set for the status of an event on a replica to indicate that the event was created on the replication source server and replicated to the replica, but is not executed on the replica. See Section 17.5.1.16, "Replication of Invoked Features".

You may supply a comment for an event using a `COMMENT` clause. *`comment`* may be any string of up to 64 characters that you wish to use for describing the event. The comment text, being a string literal, must be surrounded by quotation marks.

The `DO` clause specifies an action carried by the event, and consists of an SQL statement. Nearly any valid MySQL statement that can be used in a stored routine can also be used as the action statement for a scheduled event. (See Section 25.8, "Restrictions on Stored Programs".) For example, the following event `e_hourly` deletes all rows from the `sessions` table once per hour, where this table is part of the `site_activity` schema:

```
CREATE EVENT e_hourly
    ON SCHEDULE
      EVERY 1 HOUR
    COMMENT 'Clears out sessions table each hour.'
    DO
      DELETE FROM site_activity.sessions;
```

MySQL stores the `sql_mode` system variable setting in effect when an event is created or altered, and always executes the event with this setting in force, *regardless of the current server SQL mode when the event begins executing*.

A `CREATE EVENT` statement that contains an `ALTER EVENT` statement in its `DO` clause appears to succeed; however, when the server attempts to execute the resulting scheduled event, the execution fails with an error.

**Note**

Statements such as `SELECT` or `SHOW` that merely return a result set have no effect when used in an event; the output from these is not sent to the MySQL Monitor, nor is it stored anywhere. However, you can use statements such as `SELECT ... INTO` and `INSERT INTO ... SELECT` that store a result. (See the next example in this section for an instance of the latter.)

The schema to which an event belongs is the default schema for table references in the `DO` clause. Any references to tables in other schemas must be qualified with the proper schema name.

As with stored routines, you can use compound-statement syntax in the `DO` clause by using the `BEGIN` and `END` keywords, as shown here:

```
delimiter |

CREATE EVENT e_daily
    ON SCHEDULE
      EVERY 1 DAY
    COMMENT 'Saves total number of sessions then clears the table each day'
    DO
      BEGIN
        INSERT INTO site_activity.totals (time, total)
          SELECT CURRENT_TIMESTAMP, COUNT(*)
            FROM site_activity.sessions;
        DELETE FROM site_activity.sessions;
      END |

delimiter ;
```

This example uses the `delimiter` command to change the statement delimiter. See Section 25.1, “Defining Stored Programs”.

More complex compound statements, such as those used in stored routines, are possible in an event. This example uses local variables, an error handler, and a flow control construct:

```
delimiter |

CREATE EVENT e
    ON SCHEDULE
      EVERY 5 SECOND
    DO
      BEGIN
        DECLARE v INTEGER;
        DECLARE CONTINUE HANDLER FOR SQLEXCEPTION BEGIN END;

        SET v = 0;

        WHILE v < 5 DO
          INSERT INTO t1 VALUES (0);
          UPDATE t2 SET s1 = s1 + 1;
          SET v = v + 1;
        END WHILE;
    END |

delimiter ;
```

There is no way to pass parameters directly to or from events; however, it is possible to invoke a stored routine with parameters within an event:

```
CREATE EVENT e_call_myproc
    ON SCHEDULE
      AT CURRENT_TIMESTAMP + INTERVAL 1 DAY
    DO CALL myproc(5, 27);
```

If an event's definer has privileges sufficient to set global system variables (see Section 5.1.9.1, “System Variable Privileges”), the event can read and write global variables. As granting such privileges entails a potential for abuse, extreme care must be taken in doing so.

Generally, any statements that are valid in stored routines may be used for action statements executed by events. For more information about statements permissible within stored routines, see Section 25.2.1, “Stored Routine Syntax”. It is not possible to create an event as part of a stored routine or to create an event by another event.

## 13.1.14 CREATE FUNCTION Statement

The `CREATE FUNCTION` statement is used to create stored functions and loadable functions:

- For information about creating stored functions, see Section 13.1.17, “CREATE PROCEDURE and CREATE FUNCTION Statements”.
- For information about creating loadable functions, see Section 13.7.4.1, “CREATE FUNCTION Statement for Loadable Functions”.

## 13.1.15 CREATE INDEX Statement

```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX index_name
    [index_type]
    ON tbl_name (key_part,...)
    [index_option]
    [algorithm_option | lock_option] ...

key_part: {col_name [(length)] | (expr)} [ASC | DESC]

index_option: {
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
  | {VISIBLE | INVISIBLE}
  | ENGINE_ATTRIBUTE [=] 'string'
  | SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
}

index_type:
    USING {BTREE | HASH}

algorithm_option:
    ALGORITHM [=] {DEFAULT | INPLACE | COPY}

lock_option:
    LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE}
```

Normally, you create all indexes on a table at the time the table itself is created with `CREATE TABLE`. See Section 13.1.20, “CREATE TABLE Statement”. This guideline is especially important for `InnoDB` tables, where the primary key determines the physical layout of rows in the data file. `CREATE INDEX` enables you to add indexes to existing tables.

`CREATE INDEX` is mapped to an `ALTER TABLE` statement to create indexes. See Section 13.1.9, “ALTER TABLE Statement”. `CREATE INDEX` cannot be used to create a `PRIMARY KEY`; use `ALTER TABLE` instead. For more information about indexes, see Section 8.3.1, “How MySQL Uses Indexes”.

`InnoDB` supports secondary indexes on virtual columns. For more information, see Section 13.1.20.9, “Secondary Indexes and Generated Columns”.

When the `innodb_stats_persistent` setting is enabled, run the `ANALYZE TABLE` statement for an `InnoDB` table after creating an index on that table.

Beginning with MySQL 8.0.17, the *`expr`* for a *`key_part`* specification can take the form `(CAST` *`json_expression`* `AS` *`type`* `ARRAY)` to create a multi-valued index on a `JSON` column. See Multi-Valued Indexes.

An index specification of the form `(key_part1, key_part2, ...)` creates an index with multiple key parts. Index key values are formed by concatenating the values of the given key parts. For example `(col1, col2, col3)` specifies a multiple-column index with index keys consisting of values from `col1`, `col2`, and `col3`.

A *`key_part`* specification can end with `ASC` or `DESC` to specify whether index values are stored in ascending or descending order. The default is ascending if no order specifier is given. `ASC` and `DESC` are not permitted for `HASH` indexes. `ASC` and `DESC` are also not supported for multi-valued indexes. As of MySQL 8.0.12, `ASC` and `DESC` are not permitted for `SPATIAL` indexes.

The following sections describe different aspects of the `CREATE INDEX` statement:

- Column Prefix Key Parts
- Functional Key Parts
- Unique Indexes
- Full-Text Indexes
- Multi-Valued Indexes
- Spatial Indexes
- Index Options
- Table Copying and Locking Options

## Column Prefix Key Parts

For string columns, indexes can be created that use only the leading part of column values, using *`col_name(length)`* syntax to specify an index prefix length:

- Prefixes can be specified for `CHAR`, `VARCHAR`, `BINARY`, and `VARBINARY` key parts.
- Prefixes *must* be specified for `BLOB` and `TEXT` key parts. Additionally, `BLOB` and `TEXT` columns can be indexed only for `InnoDB`, `MyISAM`, and `BLACKHOLE` tables.
- Prefix *limits* are measured in bytes. However, prefix *lengths* for index specifications in `CREATE TABLE`, `ALTER TABLE`, and `CREATE INDEX` statements are interpreted as number of characters for nonbinary string types (`CHAR`, `VARCHAR`, `TEXT`) and number of bytes for binary string types (`BINARY`, `VARBINARY`, `BLOB`). Take this into account when specifying a prefix length for a nonbinary string column that uses a multibyte character set.

  Prefix support and lengths of prefixes (where supported) are storage engine dependent. For example, a prefix can be up to 767 bytes long for `InnoDB` tables that use the `REDUNDANT` or `COMPACT` row format. The prefix length limit is 3072 bytes for `InnoDB` tables that use the `DYNAMIC` or `COMPRESSED` row format. For `MyISAM` tables, the prefix length limit is 1000 bytes. The `NDB` storage engine does not support prefixes (see Section 23.2.7.6, “Unsupported or Missing Features in NDB Cluster”).

If a specified index prefix exceeds the maximum column data type size, `CREATE INDEX` handles the index as follows:

- For a nonunique index, either an error occurs (if strict SQL mode is enabled), or the index length is reduced to lie within the maximum column data type size and a warning is produced (if strict SQL mode is not enabled).
- For a unique index, an error occurs regardless of SQL mode because reducing the index length might enable insertion of nonunique entries that do not meet the specified uniqueness requirement.

The statement shown here creates an index using the first 10 characters of the `name` column (assuming that `name` has a nonbinary string type):

```
CREATE INDEX part_of_name ON customer (name(10));
```

If names in the column usually differ in the first 10 characters, lookups performed using this index should not be much slower than using an index created from the entire `name` column. Also, using column prefixes for indexes can make the index file much smaller, which could save a lot of disk space and might also speed up `INSERT` operations.

## Functional Key Parts

A “normal” index indexes column values or prefixes of column values. For example, in the following table, the index entry for a given `t1` row includes the full `col1` value and a prefix of the `col2` value consisting of its first 10 characters:

```
CREATE TABLE t1 (
  col1 VARCHAR(10),
  col2 VARCHAR(20),
  INDEX (col1, col2(10))
);
```

MySQL 8.0.13 and higher supports functional key parts that index expression values rather than column or column prefix values. Use of functional key parts enables indexing of values not stored directly in the table. Examples:

```
CREATE TABLE t1 (col1 INT, col2 INT, INDEX func_index ((ABS(col1))));
CREATE INDEX idx1 ON t1 ((col1 + col2));
CREATE INDEX idx2 ON t1 ((col1 + col2), (col1 - col2), col1);
ALTER TABLE t1 ADD INDEX ((col1 * 40) DESC);
```

An index with multiple key parts can mix nonfunctional and functional key parts.

`ASC` and `DESC` are supported for functional key parts.

Functional key parts must adhere to the following rules. An error occurs if a key part definition contains disallowed constructs.

- In index definitions, enclose expressions within parentheses to distinguish them from columns or column prefixes. For example, this is permitted; the expressions are enclosed within parentheses:

  ```
  INDEX ((col1 + col2), (col3 - col4))
  ```

  This produces an error; the expressions are not enclosed within parentheses:

  ```
  INDEX (col1 + col2, col3 - col4)
  ```

- A functional key part cannot consist solely of a column name. For example, this is not permitted:

  ```
  INDEX ((col1), (col2))
  ```

  Instead, write the key parts as nonfunctional key parts, without parentheses:

  ```
  INDEX (col1, col2)
  ```

- A functional key part expression cannot refer to column prefixes. For a workaround, see the discussion of `SUBSTRING()` and `CAST()` later in this section.

- Functional key parts are not permitted in foreign key specifications.

For `CREATE TABLE ... LIKE`, the destination table preserves functional key parts from the original table.

Functional indexes are implemented as hidden virtual generated columns, which has these implications:

- Each functional key part counts against the limit on total number of table columns; see Section 8.4.7, “Limits on Table Column Count and Row Size”.

- Functional key parts inherit all restrictions that apply to generated columns. Examples:
  - Only functions permitted for generated columns are permitted for functional key parts.
  - Subqueries, parameters, variables, stored functions, and loadable functions are not permitted.

  For more information about applicable restrictions, see Section 13.1.20.8, “CREATE TABLE and Generated Columns”, and Section 13.1.9.2, “ALTER TABLE and Generated Columns”.
- The virtual generated column itself requires no storage. The index itself takes up storage space as any other index.

`UNIQUE` is supported for indexes that include functional key parts. However, primary keys cannot include functional key parts. A primary key requires the generated column to be stored, but functional key parts are implemented as virtual generated columns, not stored generated columns.

`SPATIAL` and `FULLTEXT` indexes cannot have functional key parts.

If a table contains no primary key, `InnoDB` automatically promotes the first `UNIQUE NOT NULL` index to the primary key. This is not supported for `UNIQUE NOT NULL` indexes that have functional key parts.

Nonfunctional indexes raise a warning if there are duplicate indexes. Indexes that contain functional key parts do not have this feature.

To remove a column that is referenced by a functional key part, the index must be removed first. Otherwise, an error occurs.

Although nonfunctional key parts support a prefix length specification, this is not possible for functional key parts. The solution is to use `SUBSTRING()` (or `CAST()`, as described later in this section). For a functional key part containing the `SUBSTRING()` function to be used in a query, the `WHERE` clause must contain `SUBSTRING()` with the same arguments. In the following example, only the second `SELECT` is able to use the index because that is the only query in which the arguments to `SUBSTRING()` match the index specification:

```
CREATE TABLE tbl (
  col1 LONGTEXT,
  INDEX idx1 ((SUBSTRING(col1, 1, 10)))
);
SELECT * FROM tbl WHERE SUBSTRING(col1, 1, 9) = '123456789';
SELECT * FROM tbl WHERE SUBSTRING(col1, 1, 10) = '1234567890';
```

Functional key parts enable indexing of values that cannot be indexed otherwise, such as `JSON` values. However, this must be done correctly to achieve the desired effect. For example, this syntax does not work:

```
CREATE TABLE employees (
  data JSON,
  INDEX ((data->>'$.name'))
);
```

The syntax fails because:

- The `->>` operator translates into `JSON_UNQUOTE(JSON_EXTRACT(...))`.
- `JSON_UNQUOTE()` returns a value with a data type of `LONGTEXT`, and the hidden generated column thus is assigned the same data type.
- MySQL cannot index `LONGTEXT` columns specified without a prefix length on the key part, and prefix lengths are not permitted in functional key parts.

To index the `JSON` column, you could try using the `CAST()` function as follows:

```
CREATE TABLE employees (
```

```
  data JSON,
  INDEX ((CAST(data->>'$.name' AS CHAR(30))))
);
```

The hidden generated column is assigned the `VARCHAR(30)` data type, which can be indexed. But this approach produces a new issue when trying to use the index:

- `CAST()` returns a string with the collation `utf8mb4_0900_ai_ci` (the server default collation).
- `JSON_UNQUOTE()` returns a string with the collation `utf8mb4_bin` (hard coded).

As a result, there is a collation mismatch between the indexed expression in the preceding table definition and the `WHERE` clause expression in the following query:

```
SELECT * FROM employees WHERE data->>'$.name' = 'James';
```

The index is not used because the expressions in the query and the index differ. To support this kind of scenario for functional key parts, the optimizer automatically strips `CAST()` when looking for an index to use, but *only* if the collation of the indexed expression matches that of the query expression. For an index with a functional key part to be used, either of the following two solutions work (although they differ somewhat in effect):

- Solution 1. Assign the indexed expression the same collation as `JSON_UNQUOTE()`:

  ```
  CREATE TABLE employees (
    data JSON,
    INDEX idx ((CAST(data->>"$.name" AS CHAR(30)) COLLATE utf8mb4_bin))
  );
  INSERT INTO employees VALUES
    ('{ "name": "james", "salary": 9000 }'),
    ('{ "name": "James", "salary": 10000 }'),
    ('{ "name": "Mary", "salary": 12000 }'),
    ('{ "name": "Peter", "salary": 8000 }');
  SELECT * FROM employees WHERE data->>'$.name' = 'James';
  ```

  The `->>` operator is the same as `JSON_UNQUOTE(JSON_EXTRACT(...))`, and `JSON_UNQUOTE()` returns a string with collation `utf8mb4_bin`. The comparison is thus case-sensitive, and only one row matches:

  ```
  +----------------------------------+
  | data                             |
  +----------------------------------+
  | {"name": "James", "salary": 10000} |
  +----------------------------------+
  ```

- Solution 2. Specify the full expression in the query:

  ```
  CREATE TABLE employees (
    data JSON,
    INDEX idx ((CAST(data->>"$.name" AS CHAR(30))))
  );
  INSERT INTO employees VALUES
    ('{ "name": "james", "salary": 9000 }'),
    ('{ "name": "James", "salary": 10000 }'),
    ('{ "name": "Mary", "salary": 12000 }'),
    ('{ "name": "Peter", "salary": 8000 }');
  SELECT * FROM employees WHERE CAST(data->>'$.name' AS CHAR(30)) = 'James';
  ```

  `CAST()` returns a string with collation `utf8mb4_0900_ai_ci`, so the comparison case-insensitive and two rows match:

  ```
  +----------------------------------+
  | data                             |
  +----------------------------------+
  | {"name": "james", "salary": 9000}  |
  | {"name": "James", "salary": 10000} |
  +----------------------------------+
  ```

Be aware that although the optimizer supports automatically stripping `CAST()` with indexed generated columns, the following approach does not work because it produces a different result with and without an index (Bug#27337092):

```
mysql> CREATE TABLE employees (
         data JSON,
         generated_col VARCHAR(30) AS (CAST(data->>'$.name' AS CHAR(30)))
       );
Query OK, 0 rows affected, 1 warning (0.03 sec)

mysql> INSERT INTO employees (data)
       VALUES ('{"name": "james"}'), ('{"name": "James"}');
Query OK, 2 rows affected, 1 warning (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 1

mysql> SELECT * FROM employees WHERE data->>'$.name' = 'James';
+------------------+---------------+
| data             | generated_col |
+------------------+---------------+
| {"name": "James"} | James         |
+------------------+---------------+
1 row in set (0.00 sec)

mysql> ALTER TABLE employees ADD INDEX idx (generated_col);
Query OK, 0 rows affected, 1 warning (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 1

mysql> SELECT * FROM employees WHERE data->>'$.name' = 'James';
+------------------+---------------+
| data             | generated_col |
+------------------+---------------+
| {"name": "james"} | james         |
| {"name": "James"} | James         |
+------------------+---------------+
2 rows in set (0.01 sec)
```

## Unique Indexes

A `UNIQUE` index creates a constraint such that all values in the index must be distinct. An error occurs if you try to add a new row with a key value that matches an existing row. If you specify a prefix value for a column in a `UNIQUE` index, the column values must be unique within the prefix length. A `UNIQUE` index permits multiple `NULL` values for columns that can contain `NULL`.

If a table has a `PRIMARY KEY` or `UNIQUE NOT NULL` index that consists of a single column that has an integer type, you can use `_rowid` to refer to the indexed column in `SELECT` statements, as follows:

- `_rowid` refers to the `PRIMARY KEY` column if there is a `PRIMARY KEY` consisting of a single integer column. If there is a `PRIMARY KEY` but it does not consist of a single integer column, `_rowid` cannot be used.
- Otherwise, `_rowid` refers to the column in the first `UNIQUE NOT NULL` index if that index consists of a single integer column. If the first `UNIQUE NOT NULL` index does not consist of a single integer column, `_rowid` cannot be used.

## Full-Text Indexes

`FULLTEXT` indexes are supported only for `InnoDB` and `MyISAM` tables and can include only `CHAR`, `VARCHAR`, and `TEXT` columns. Indexing always happens over the entire column; column prefix indexing is not supported and any prefix length is ignored if specified. See Section 12.10, “Full-Text Search Functions”, for details of operation.

## Multi-Valued Indexes

As of MySQL 8.0.17, `InnoDB` supports multi-valued indexes. A multi-valued index is a secondary index defined on a column that stores an array of values. A “normal” index has one index record for each data record (1:1). A multi-valued index can have multiple index records for a single data record (N:1).

Multi-valued indexes are intended for indexing `JSON` arrays. For example, a multi-valued index defined on the array of zip codes in the following JSON document creates an index record for each zip code, with each index record referencing the same data record.

```
{
    "user":"Bob",
    "user_id":31,
    "zipcode":[94477,94536]
}
```

### Creating multi-valued Indexes

You can create a multi-valued index in a `CREATE TABLE`, `ALTER TABLE`, or `CREATE INDEX` statement. This requires using `CAST(... AS ... ARRAY)` in the index definition, which casts same-typed scalar values in a `JSON` array to an SQL data type array. A virtual column is then generated transparently with the values in the SQL data type array; finally, a functional index (also referred to as a virtual index) is created on the virtual column. It is the functional index defined on the virtual column of values from the SQL data type array that forms the multi-valued index.

The examples in the following list show the three different ways in which a multi-valued index `zips` can be created on an array `$.zipcode` on a `JSON` column `custinfo` in a table named `customers`. In each case, the JSON array is cast to an SQL data type array of `UNSIGNED` integer values.

- `CREATE TABLE` only:

```
CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    modified DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    custinfo JSON,
    INDEX zips( (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)) )
    );
```

- `CREATE TABLE` plus `ALTER TABLE`:

```
CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    modified DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    custinfo JSON
    );

ALTER TABLE customers ADD INDEX zips( (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)) );
```

- `CREATE TABLE` plus `CREATE INDEX`:

```
CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    modified DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    custinfo JSON
    );

CREATE INDEX zips ON customers ( (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)) );
```

A multi-valued index can also be defined as part of a composite index. This example shows a composite index that includes two single-valued parts (for the `id` and `modified` columns), and one multi-valued part (for the `custinfo` column):

```
CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    modified DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    custinfo JSON
    );

ALTER TABLE customers ADD INDEX comp(id, modified,
    (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)) );
```

Only one multi-valued key part can be used in a composite index. The multi-valued key part may be used in any order relative to the other parts of the key. In other words, the `ALTER TABLE` statement

just shown could have used `comp(id, (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY), modified))` (or any other ordering) and still have been valid.

### Using multi-valued Indexes

The optimizer uses a multi-valued index to fetch records when the following functions are specified in a `WHERE` clause:

- `MEMBER OF()`
- `JSON_CONTAINS()`
- `JSON_OVERLAPS()`

We can demonstrate this by creating and populating the `customers` table using the following `CREATE TABLE` and `INSERT` statements:

```
mysql> CREATE TABLE customers (
    ->     id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->     modified DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    ->     custinfo JSON
    ->     );
Query OK, 0 rows affected (0.51 sec)

mysql> INSERT INTO customers VALUES
    ->     (NULL, NOW(), '{"user":"Jack","user_id":37,"zipcode":[94582,94536]}'),
    ->     (NULL, NOW(), '{"user":"Jill","user_id":22,"zipcode":[94568,94507,94582]}'),
    ->     (NULL, NOW(), '{"user":"Bob","user_id":31,"zipcode":[94477,94507]}'),
    ->     (NULL, NOW(), '{"user":"Mary","user_id":72,"zipcode":[94536]}'),
    ->     (NULL, NOW(), '{"user":"Ted","user_id":56,"zipcode":[94507,94582]}');
Query OK, 5 rows affected (0.07 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

First we execute three queries on the `customers` table, one each using `MEMBER OF()`, `JSON_CONTAINS()`, and `JSON_OVERLAPS()`, with the result from each query shown here:

```
mysql> SELECT * FROM customers
    ->     WHERE 94507 MEMBER OF(custinfo->'$.zipcode');
+----+---------------------+------------------------------------------------------------------------+
| id | modified            | custinfo                                                               |
+----+---------------------+------------------------------------------------------------------------+
|  2 | 2019-06-29 22:23:12 | {"user": "Jill", "user_id": 22, "zipcode": [94568, 94507, 94582]} |
|  3 | 2019-06-29 22:23:12 | {"user": "Bob", "user_id": 31, "zipcode": [94477, 94507]}         |
|  5 | 2019-06-29 22:23:12 | {"user": "Ted", "user_id": 56, "zipcode": [94507, 94582]}         |
+----+---------------------+------------------------------------------------------------------------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM customers
    ->     WHERE JSON_CONTAINS(custinfo->'$.zipcode', CAST('[94507,94582]' AS JSON));
+----+---------------------+------------------------------------------------------------------------+
| id | modified            | custinfo                                                               |
+----+---------------------+------------------------------------------------------------------------+
|  2 | 2019-06-29 22:23:12 | {"user": "Jill", "user_id": 22, "zipcode": [94568, 94507, 94582]} |
|  5 | 2019-06-29 22:23:12 | {"user": "Ted", "user_id": 56, "zipcode": [94507, 94582]}         |
+----+---------------------+------------------------------------------------------------------------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM customers
    ->     WHERE JSON_OVERLAPS(custinfo->'$.zipcode', CAST('[94507,94582]' AS JSON));
+----+---------------------+------------------------------------------------------------------------+
| id | modified            | custinfo                                                               |
+----+---------------------+------------------------------------------------------------------------+
|  1 | 2019-06-29 22:23:12 | {"user": "Jack", "user_id": 37, "zipcode": [94582, 94536]}        |
|  2 | 2019-06-29 22:23:12 | {"user": "Jill", "user_id": 22, "zipcode": [94568, 94507, 94582]} |
|  3 | 2019-06-29 22:23:12 | {"user": "Bob", "user_id": 31, "zipcode": [94477, 94507]}         |
|  5 | 2019-06-29 22:23:12 | {"user": "Ted", "user_id": 56, "zipcode": [94507, 94582]}         |
+----+---------------------+------------------------------------------------------------------------+
4 rows in set (0.00 sec)
```

Next, we run `EXPLAIN` on each of the previous three queries:

```
mysql> EXPLAIN SELECT * FROM customers
    ->     WHERE 94507 MEMBER OF(custinfo->'$.zipcode');
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+--------
| id | select_type | table     | partitions | type | possible_keys | key  | key_len | ref  | rows | filtere
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+--------
|  1 | SIMPLE      | customers | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    5 |   100.0
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+--------
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM customers
    ->     WHERE JSON_CONTAINS(custinfo->'$.zipcode', CAST('[94507,94582]' AS JSON));
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+--------
| id | select_type | table     | partitions | type | possible_keys | key  | key_len | ref  | rows | filtere
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+--------
|  1 | SIMPLE      | customers | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    5 |   100.0
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+--------
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM customers
    ->     WHERE JSON_OVERLAPS(custinfo->'$.zipcode', CAST('[94507,94582]' AS JSON));
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+--------
| id | select_type | table     | partitions | type | possible_keys | key  | key_len | ref  | rows | filtere
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+--------
|  1 | SIMPLE      | customers | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    5 |   100.0
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+--------
1 row in set, 1 warning (0.01 sec)
```

None of the three queries just shown are able to use any keys. To solve this problem, we can add a multi-valued index on the `zipcode` array in the `JSON` column (`custinfo`), like this:

```
mysql> ALTER TABLE customers
    ->     ADD INDEX zips( (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)) );
Query OK, 0 rows affected (0.47 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

When we run the previous `EXPLAIN` statements again, we can now observe that the queries can (and do) use the index `zips` that was just created:

```
mysql> EXPLAIN SELECT * FROM customers
    ->     WHERE 94507 MEMBER OF(custinfo->'$.zipcode');
+----+-------------+-----------+------------+------+---------------+------+---------+-------+------+-------
| id | select_type | table     | partitions | type | possible_keys | key  | key_len | ref   | rows | filter
+----+-------------+-----------+------------+------+---------------+------+---------+-------+------+-------
|  1 | SIMPLE      | customers | NULL       | ref  | zips          | zips | 9       | const |    1 |   100.
+----+-------------+-----------+------------+------+---------------+------+---------+-------+------+-------
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM customers
    ->     WHERE JSON_CONTAINS(custinfo->'$.zipcode', CAST('[94507,94582]' AS JSON));
+----+-------------+-----------+------------+-------+---------------+------+---------+------+------+-------
| id | select_type | table     | partitions | type  | possible_keys | key  | key_len | ref  | rows | filter
+----+-------------+-----------+------------+-------+---------------+------+---------+------+------+-------
|  1 | SIMPLE      | customers | NULL       | range | zips          | zips | 9       | NULL |    6 |   100.
+----+-------------+-----------+------------+-------+---------------+------+---------+------+------+-------
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM customers
    ->     WHERE JSON_OVERLAPS(custinfo->'$.zipcode', CAST('[94507,94582]' AS JSON));
+----+-------------+-----------+------------+-------+---------------+------+---------+------+------+-------
| id | select_type | table     | partitions | type  | possible_keys | key  | key_len | ref  | rows | filter
+----+-------------+-----------+------------+-------+---------------+------+---------+------+------+-------
|  1 | SIMPLE      | customers | NULL       | range | zips          | zips | 9       | NULL |    6 |   100.
+----+-------------+-----------+------------+-------+---------------+------+---------+------+------+-------
1 row in set, 1 warning (0.01 sec)
```

A multi-valued index can be defined as a unique key. If defined as a unique key, attempting to insert a value already present in the multi-valued index returns a duplicate key error. If duplicate values are already present, attempting to add a unique multi-valued index fails, as shown here:

```
mysql> ALTER TABLE customers DROP INDEX zips;
Query OK, 0 rows affected (0.55 sec)
```

```
Records: 0  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE customers
    ->     ADD UNIQUE INDEX zips((CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)));
ERROR 1062 (23000): Duplicate entry '[94507, ' for key 'customers.zips'
mysql> ALTER TABLE customers
    ->     ADD INDEX zips((CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)));
Query OK, 0 rows affected (0.36 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

### Characteristics of Multi-Valued Indexes

Multi-valued indexes have the additional characteristics listed here:

- DML operations that affect multi-valued indexes are handled in the same way as DML operations that affect a normal index, with the only difference being that there may be more than one insert or update for a single clustered index record.
- Nullability and multi-valued indexes:
  - If a multi-valued key part has an empty array, no entries are added to the index, and the data record is not accessible by an index scan.
  - If multi-valued key part generation returns a `NULL` value, a single entry containing `NULL` is added to the multi-valued index. If the key part is defined as `NOT NULL`, an error is reported.
  - If the typed array column is set to `NULL`, the storage engine stores a single record containing `NULL` that points to the data record.
  - `JSON` null values are not permitted in indexed arrays. If any returned value is `NULL`, it is treated as a JSON null and an `Invalid JSON value` error is reported.
- Because multi-valued indexes are virtual indexes on virtual columns, they must adhere to the same rules as secondary indexes on virtual generated columns.
- Index records are not added for empty arrays.

### Limitations and Restrictions on Multi-valued Indexes

Multi-valued indexes are subject to the limitations and restrictions listed here:

- Only one multi-valued key part is permitted per multi-valued index. However, the `CAST( ... AS ... ARRAY)` expression can refer to multiple arrays within a `JSON` document, as shown here:

  ```
  CAST(data->'$.arr[*][*]' AS UNSIGNED ARRAY)
  ```

  In this case, all values matching the JSON expression are stored in the index as a single flat array.
- An index with a multi-valued key part does not support ordering and therefore cannot be used as a primary key. For the same reason, a multi-valued index cannot be defined using the `ASC` or `DESC` keyword.
- A multi-valued index cannot be a covering index.
- The maximum number of values per record for a multi-valued index is determined by the amount of data than can be stored on a single undo log page, which is 65221 bytes (64K minus 315 bytes for overhead), which means that the maximum total length of key values is also 65221 bytes. The maximum number of keys depends on various factors, which prevents defining a specific limit. Tests have shown a multi-valued index to permit as many as 1604 integer keys per record, for example. When the limit is reached, an error similar to the following is reported: `ERROR 3905 (HY000): Exceeded max number of values per record for multi-valued index 'idx' by 1 value(s).`

- The only type of expression that is permitted in a multi-valued key part is a `JSON` expression. The expression need not reference an existing element in a JSON document inserted into the indexed column, but must itself be syntactically valid.
- Because index records for the same clustered index record are dispersed throughout a multi-valued index, a multi-valued index does not support range scans or index-only scans.
- Multi-valued indexes are not permitted in foreign key specifications.
- Index prefixes cannot be defined for multi-valued indexes.
- Multi-valued indexes cannot be defined on data cast as `BINARY` (see the description of the `CAST()` function).
- Online creation of a multi-value index is not supported, which means the operation uses `ALGORITHM=COPY`. See Performance and Space Requirements.
- Character sets and collations other than the following two combinations of character set and collation are not supported for multi-valued indexes:
  1. The `binary` character set with the default `binary` collation
  2. The `utf8mb4` character set with the default `utf8mb4_0900_as_cs` collation.
- As with other indexes on columns of `InnoDB` tables, a multi-valued index cannot be created with `USING HASH`; attempting to do so results in a warning: `This storage engine does not support the HASH index algorithm, storage engine default was used instead`. (`USING BTREE` is supported as usual.)

## Spatial Indexes

The `MyISAM`, `InnoDB`, `NDB`, and `ARCHIVE` storage engines support spatial columns such as `POINT` and `GEOMETRY`. (Section 11.4, “Spatial Data Types”, describes the spatial data types.) However, support for spatial column indexing varies among engines. Spatial and nonspatial indexes on spatial columns are available according to the following rules.

Spatial indexes on spatial columns have these characteristics:

- Available only for `InnoDB` and `MyISAM` tables. Specifying `SPATIAL INDEX` for other storage engines results in an error.
- As of MySQL 8.0.12, an index on a spatial column *must* be a `SPATIAL` index. The `SPATIAL` keyword is thus optional but implicit for creating an index on a spatial column.
- Available for single spatial columns only. A spatial index cannot be created over multiple spatial columns.
- Indexed columns must be `NOT NULL`.
- Column prefix lengths are prohibited. The full width of each column is indexed.
- Not permitted for a primary key or unique index.

Nonspatial indexes on spatial columns (created with `INDEX`, `UNIQUE`, or `PRIMARY KEY`) have these characteristics:

- Permitted for any storage engine that supports spatial columns except `ARCHIVE`.
- Columns can be `NULL` unless the index is a primary key.
- The index type for a non-`SPATIAL` index depends on the storage engine. Currently, B-tree is used.
- Permitted for a column that can have `NULL` values only for `InnoDB`, `MyISAM`, and `MEMORY` tables.

## Index Options

Following the key part list, index options can be given. An *index_option* value can be any of the following:

- KEY_BLOCK_SIZE [=] *value*

  For MyISAM tables, KEY_BLOCK_SIZE optionally specifies the size in bytes to use for index key blocks. The value is treated as a hint; a different size could be used if necessary. A KEY_BLOCK_SIZE value specified for an individual index definition overrides a table-level KEY_BLOCK_SIZE value.

  KEY_BLOCK_SIZE is not supported at the index level for InnoDB tables. See Section 13.1.20, "CREATE TABLE Statement".

- *index_type*

  Some storage engines permit you to specify an index type when creating an index. For example:

```
CREATE TABLE lookup (id INT) ENGINE = MEMORY;
CREATE INDEX id_index ON lookup (id) USING BTREE;
```

  Table 13.1, "Index Types Per Storage Engine" shows the permissible index type values supported by different storage engines. Where multiple index types are listed, the first one is the default when no index type specifier is given. Storage engines not listed in the table do not support an *index_type* clause in index definitions.

**Table 13.1 Index Types Per Storage Engine**

| Storage Engine | Permissible Index Types |
|---|---|
| InnoDB | BTREE |
| MyISAM | BTREE |
| MEMORY/HEAP | HASH, BTREE |
| NDB | HASH, BTREE (see note in text) |

The *index_type* clause cannot be used for FULLTEXT INDEX or (prior to MySQL 8.0.12) SPATIAL INDEX specifications. Full-text index implementation is storage engine dependent. Spatial indexes are implemented as R-tree indexes.

If you specify an index type that is not valid for a given storage engine, but another index type is available that the engine can use without affecting query results, the engine uses the available type. The parser recognizes RTREE as a type name. As of MySQL 8.0.12, this is permitted only for SPATIAL indexes. Prior to 8.0.12, RTREE cannot be specified for any storage engine.

BTREE indexes are implemented by the NDB storage engine as T-tree indexes.

**Note**

For indexes on NDB table columns, the USING option can be specified only for a unique index or primary key. USING HASH prevents the creation of an ordered index; otherwise, creating a unique index or primary key on an NDB table automatically results in the creation of both an ordered index and a hash index, each of which indexes the same set of columns.

For unique indexes that include one or more NULL columns of an NDB table, the hash index can be used only to look up literal values, which means that IS [NOT] NULL conditions require a full scan of the table. One workaround is to make sure that a unique index using one or more NULL columns on such

a table is always created in such a way that it includes the ordered index; that is, avoid employing `USING HASH` when creating the index.

If you specify an index type that is not valid for a given storage engine, but another index type is available that the engine can use without affecting query results, the engine uses the available type. The parser recognizes `RTREE` as a type name, but currently this cannot be specified for any storage engine.

**Note**

Use of the *`index_type`* option before the `ON tbl_name` clause is deprecated; expect support for use of the option in this position to be removed in a future MySQL release. If an *`index_type`* option is given in both the earlier and later positions, the final option applies.

`TYPE type_name` is recognized as a synonym for `USING type_name`. However, `USING` is the preferred form.

The following tables show index characteristics for the storage engines that support the *`index_type`* option.

**Table 13.2 InnoDB Storage Engine Index Characteristics**

| Index Class | Index Type | Stores NULL VALUES | Permits Multiple NULL Values | IS NULL Scan Type | IS NOT NULL Scan Type |
|---|---|---|---|---|---|
| Primary key | `BTREE` | No | No | N/A | N/A |
| Unique | `BTREE` | Yes | Yes | Index | Index |
| Key | `BTREE` | Yes | Yes | Index | Index |
| `FULLTEXT` | N/A | Yes | Yes | Table | Table |
| `SPATIAL` | N/A | No | No | N/A | N/A |

**Table 13.3 MyISAM Storage Engine Index Characteristics**

| Index Class | Index Type | Stores NULL VALUES | Permits Multiple NULL Values | IS NULL Scan Type | IS NOT NULL Scan Type |
|---|---|---|---|---|---|
| Primary key | `BTREE` | No | No | N/A | N/A |
| Unique | `BTREE` | Yes | Yes | Index | Index |
| Key | `BTREE` | Yes | Yes | Index | Index |
| `FULLTEXT` | N/A | Yes | Yes | Table | Table |
| `SPATIAL` | N/A | No | No | N/A | N/A |

**Table 13.4 MEMORY Storage Engine Index Characteristics**

| Index Class | Index Type | Stores NULL VALUES | Permits Multiple NULL Values | IS NULL Scan Type | IS NOT NULL Scan Type |
|---|---|---|---|---|---|
| Primary key | `BTREE` | No | No | N/A | N/A |
| Unique | `BTREE` | Yes | Yes | Index | Index |
| Key | `BTREE` | Yes | Yes | Index | Index |
| Primary key | `HASH` | No | No | N/A | N/A |
| Unique | `HASH` | Yes | Yes | Index | Index |

| Index Class | Index Type | Stores NULL VALUES | Permits Multiple NULL Values | IS NULL Scan Type | IS NOT NULL Scan Type |
|---|---|---|---|---|---|
| Key | HASH | Yes | Yes | Index | Index |

**Table 13.5 NDB Storage Engine Index Characteristics**

| Index Class | Index Type | Stores NULL VALUES | Permits Multiple NULL Values | IS NULL Scan Type | IS NOT NULL Scan Type |
|---|---|---|---|---|---|
| Primary key | BTREE | No | No | Index | Index |
| Unique | BTREE | Yes | Yes | Index | Index |
| Key | BTREE | Yes | Yes | Index | Index |
| Primary key | HASH | No | No | Table (see note 1) | Table (see note 1) |
| Unique | HASH | Yes | Yes | Table (see note 1) | Table (see note 1) |
| Key | HASH | Yes | Yes | Table (see note 1) | Table (see note 1) |

Table note:

1. `USING HASH` prevents creation of an implicit ordered index.

- `WITH PARSER` *`parser_name`*

  This option can be used only with `FULLTEXT` indexes. It associates a parser plugin with the index if full-text indexing and searching operations need special handling. `InnoDB` and `MyISAM` support full-text parser plugins. If you have a `MyISAM` table with an associated full-text parser plugin, you can convert the table to `InnoDB` using `ALTER TABLE`. See Full-Text Parser Plugins and Writing Full-Text Parser Plugins for more information.

- `COMMENT '`*`string`*`'`

  Index definitions can include an optional comment of up to 1024 characters.

  The `MERGE_THRESHOLD` for index pages can be configured for individual indexes using the *`index_option`* `COMMENT` clause of the `CREATE INDEX` statement. For example:

```
CREATE TABLE t1 (id INT);
CREATE INDEX id_index ON t1 (id) COMMENT 'MERGE_THRESHOLD=40';
```

  If the page-full percentage for an index page falls below the `MERGE_THRESHOLD` value when a row is deleted or when a row is shortened by an update operation, `InnoDB` attempts to merge the index page with a neighboring index page. The default `MERGE_THRESHOLD` value is 50, which is the previously hardcoded value.

  `MERGE_THRESHOLD` can also be defined at the index level and table level using `CREATE TABLE` and `ALTER TABLE` statements. For more information, see Section 15.8.11, “Configuring the Merge Threshold for Index Pages”.

- `VISIBLE`, `INVISIBLE`

  Specify index visibility. Indexes are visible by default. An invisible index is not used by the optimizer. Specification of index visibility applies to indexes other than primary keys (either explicit or implicit). For more information, see Section 8.3.12, “Invisible Indexes”.

- `ENGINE_ATTRIBUTE` and `SECONDARY_ENGINE_ATTRIBUTE` options (available as of MySQL 8.0.21) are used to specify index attributes for primary and secondary storage engines. The options are reserved for future use.

  Permitted values are a string literal containing a valid `JSON` document or an empty string (''). Invalid `JSON` is rejected.

  ```
  CREATE INDEX i1 ON t1 (c1) ENGINE_ATTRIBUTE='{"key":"value"}';
  ```

  `ENGINE_ATTRIBUTE` and `SECONDARY_ENGINE_ATTRIBUTE` values can be repeated without error. In this case, the last specified value is used.

  `ENGINE_ATTRIBUTE` and `SECONDARY_ENGINE_ATTRIBUTE` values are not checked by the server, nor are they cleared when the table's storage engine is changed.

### Table Copying and Locking Options

`ALGORITHM` and `LOCK` clauses may be given to influence the table copying method and level of concurrency for reading and writing the table while its indexes are being modified. They have the same meaning as for the `ALTER TABLE` statement. For more information, see Section 13.1.9, "ALTER TABLE Statement"

NDB Cluster supports online operations using the same `ALGORITHM=INPLACE` syntax used with the standard MySQL Server. See Section 23.6.12, "Online Operations with ALTER TABLE in NDB Cluster", for more information.

## 13.1.16 CREATE LOGFILE GROUP Statement

```
CREATE LOGFILE GROUP logfile_group
    ADD UNDOFILE 'undo_file'
    [INITIAL_SIZE [=] initial_size]
    [UNDO_BUFFER_SIZE [=] undo_buffer_size]
    [REDO_BUFFER_SIZE [=] redo_buffer_size]
    [NODEGROUP [=] nodegroup_id]
    [WAIT]
    [COMMENT [=] 'string']
    ENGINE [=] engine_name
```

This statement creates a new log file group named *`logfile_group`* having a single `UNDO` file named '*`undo_file`*'. A `CREATE LOGFILE GROUP` statement has one and only one `ADD UNDOFILE` clause. For rules covering the naming of log file groups, see Section 9.2, "Schema Object Names".

**Note**

All NDB Cluster Disk Data objects share the same namespace. This means that *each Disk Data object* must be uniquely named (and not merely each Disk Data object of a given type). For example, you cannot have a tablespace and a log file group with the same name, or a tablespace and a data file with the same name.

There can be only one log file group per NDB Cluster instance at any given time.

The optional `INITIAL_SIZE` parameter sets the `UNDO` file's initial size; if not specified, it defaults to `128M` (128 megabytes). The optional `UNDO_BUFFER_SIZE` parameter sets the size used by the `UNDO` buffer for the log file group; The default value for `UNDO_BUFFER_SIZE` is `8M` (eight megabytes); this value cannot exceed the amount of system memory available. Both of these parameters are specified in bytes. You may optionally follow either or both of these with a one-letter abbreviation for an order of magnitude, similar to those used in `my.cnf`. Generally, this is one of the letters `M` (for megabytes) or `G` (for gigabytes).

Memory used for `UNDO_BUFFER_SIZE` comes from the global pool whose size is determined by the value of the `SharedGlobalMemory` data node configuration parameter. This includes any default

value implied for this option by the setting of the `InitialLogFileGroup` data node configuration parameter.

The maximum permitted for `UNDO_BUFFER_SIZE` is 629145600 (600 MB).

On 32-bit systems, the maximum supported value for `INITIAL_SIZE` is 4294967296 (4 GB). (Bug #29186)

The minimum allowed value for `INITIAL_SIZE` is 1048576 (1 MB).

The `ENGINE` option determines the storage engine to be used by this log file group, with *`engine_name`* being the name of the storage engine. In MySQL 8.0, this must be `NDB` (or `NDBCLUSTER`). If `ENGINE` is not set, MySQL tries to use the engine specified by the `default_storage_engine` server system variable (formerly `storage_engine`). In any case, if the engine is not specified as `NDB` or `NDBCLUSTER`, the `CREATE LOGFILE GROUP` statement appears to succeed but actually fails to create the log file group, as shown here:

```
mysql> CREATE LOGFILE GROUP lg1
    ->     ADD UNDOFILE 'undo.dat' INITIAL_SIZE = 10M;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+-------+------+-----------------------------------------------------------------------------------------------
| Level | Code | Message
+-------+------+-----------------------------------------------------------------------------------------------
| Error | 1478 | Table storage engine 'InnoDB' does not support the create option 'TABLESPACE or LOGFIL
+-------+------+-----------------------------------------------------------------------------------------------
1 row in set (0.00 sec)

mysql> DROP LOGFILE GROUP lg1 ENGINE = NDB;
ERROR 1529 (HY000): Failed to drop LOGFILE GROUP

mysql> CREATE LOGFILE GROUP lg1
    ->     ADD UNDOFILE 'undo.dat' INITIAL_SIZE = 10M
    ->     ENGINE = NDB;
Query OK, 0 rows affected (2.97 sec)
```

The fact that the `CREATE LOGFILE GROUP` statement does not actually return an error when a non-`NDB` storage engine is named, but rather appears to succeed, is a known issue which we hope to address in a future release of NDB Cluster.

*`REDO_BUFFER_SIZE`*, `NODEGROUP`, `WAIT`, and `COMMENT` are parsed but ignored, and so have no effect in MySQL 8.0. These options are intended for future expansion.

When used with `ENGINE [=] NDB`, a log file group and associated `UNDO` log file are created on each Cluster data node. You can verify that the `UNDO` files were created and obtain information about them by querying the Information Schema `FILES` table. For example:

```
mysql> SELECT LOGFILE_GROUP_NAME, LOGFILE_GROUP_NUMBER, EXTRA
    -> FROM INFORMATION_SCHEMA.FILES
    -> WHERE FILE_NAME = 'undo_10.dat';
+--------------------+----------------------+----------------+
| LOGFILE_GROUP_NAME | LOGFILE_GROUP_NUMBER | EXTRA          |
+--------------------+----------------------+----------------+
| lg_3               |                   11 | CLUSTER_NODE=3 |
| lg_3               |                   11 | CLUSTER_NODE=4 |
+--------------------+----------------------+----------------+
2 rows in set (0.06 sec)
```

`CREATE LOGFILE GROUP` is useful only with Disk Data storage for NDB Cluster. See Section 23.6.11, “NDB Cluster Disk Data Tables”.

## 13.1.17 CREATE PROCEDURE and CREATE FUNCTION Statements

```
CREATE
    [DEFINER = user]
    PROCEDURE [IF NOT EXISTS] sp_name ([proc_parameter[,...]])
```

```
    [characteristic ...] routine_body

CREATE
    [DEFINER = user]
    FUNCTION [IF NOT EXISTS] sp_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

func_parameter:
    param_name type

type:
    Any valid MySQL data type

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:
    Valid SQL routine statement
```

These statements are used to create a stored routine (a stored procedure or function). That is, the specified routine becomes known to the server. By default, a stored routine is associated with the default database. To associate the routine explicitly with a given database, specify the name as *`db_name.sp_name`* when you create it.

The `CREATE FUNCTION` statement is also used in MySQL to support loadable functions. See Section 13.7.4.1, “CREATE FUNCTION Statement for Loadable Functions”. A loadable function can be regarded as an external stored function. Stored functions share their namespace with loadable functions. See Section 9.2.5, “Function Name Parsing and Resolution”, for the rules describing how the server interprets references to different kinds of functions.

To invoke a stored procedure, use the `CALL` statement (see Section 13.2.1, “CALL Statement”). To invoke a stored function, refer to it in an expression. The function returns a value during expression evaluation.

`CREATE PROCEDURE` and `CREATE FUNCTION` require the `CREATE ROUTINE` privilege. If the `DEFINER` clause is present, the privileges required depend on the *`user`* value, as discussed in Section 25.6, “Stored Object Access Control”. If binary logging is enabled, `CREATE FUNCTION` might require the `SUPER` privilege, as discussed in Section 25.7, “Stored Program Binary Logging”.

By default, MySQL automatically grants the `ALTER ROUTINE` and `EXECUTE` privileges to the routine creator. This behavior can be changed by disabling the `automatic_sp_privileges` system variable. See Section 25.2.2, “Stored Routines and MySQL Privileges”.

The `DEFINER` and `SQL SECURITY` clauses specify the security context to be used when checking access privileges at routine execution time, as described later in this section.

If the routine name is the same as the name of a built-in SQL function, a syntax error occurs unless you use a space between the name and the following parenthesis when defining the routine or invoking it later. For this reason, avoid using the names of existing SQL functions for your own stored routines.

The `IGNORE_SPACE` SQL mode applies to built-in functions, not to stored routines. It is always permissible to have spaces after a stored routine name, regardless of whether `IGNORE_SPACE` is enabled.

`IF NOT EXISTS` prevents an error from occurring if there already exists a routine with the same name. This option is supported with both `CREATE FUNCTION` and `CREATE PROCEDURE` beginning with MySQL 8.0.29.

If a built-in function with the same name already exists, attempting to create a stored function with `CREATE FUNCTION ... IF NOT EXISTS` succeeds with a warning indicating that it has the same name as a native function; this is no different than when performing the same `CREATE FUNCTION` statement without specifying `IF NOT EXISTS`.

If a loadable function with the same name already exists, attempting to create a stored function using `IF NOT EXISTS` succeeds with a warning. This is the same as without specifying `IF NOT EXISTS`.

See Function Name Resolution, for more information.

The parameter list enclosed within parentheses must always be present. If there are no parameters, an empty parameter list of `()` should be used. Parameter names are not case-sensitive.

Each parameter is an `IN` parameter by default. To specify otherwise for a parameter, use the keyword `OUT` or `INOUT` before the parameter name.

**Note**

Specifying a parameter as `IN`, `OUT`, or `INOUT` is valid only for a `PROCEDURE`. For a `FUNCTION`, parameters are always regarded as `IN` parameters.

An `IN` parameter passes a value into a procedure. The procedure might modify the value, but the modification is not visible to the caller when the procedure returns. An `OUT` parameter passes a value from the procedure back to the caller. Its initial value is `NULL` within the procedure, and its value is visible to the caller when the procedure returns. An `INOUT` parameter is initialized by the caller, can be modified by the procedure, and any change made by the procedure is visible to the caller when the procedure returns.

For each `OUT` or `INOUT` parameter, pass a user-defined variable in the `CALL` statement that invokes the procedure so that you can obtain its value when the procedure returns. If you are calling the procedure from within another stored procedure or function, you can also pass a routine parameter or local routine variable as an `OUT` or `INOUT` parameter. If you are calling the procedure from within a trigger, you can also pass `NEW.col_name` as an `OUT` or `INOUT` parameter.

For information about the effect of unhandled conditions on procedure parameters, see Section 13.6.7.8, “Condition Handling and OUT or INOUT Parameters”.

Routine parameters cannot be referenced in statements prepared within the routine; see Section 25.8, “Restrictions on Stored Programs”.

The following example shows a simple stored procedure that, given a country code, counts the number of cities for that country that appear in the `city` table of the `world` database. The country code is passed using an `IN` parameter, and the city count is returned using an `OUT` parameter:

```
mysql> delimiter //

mysql> CREATE PROCEDURE citycount (IN country CHAR(3), OUT cities INT)
       BEGIN
         SELECT COUNT(*) INTO cities FROM world.city
         WHERE CountryCode = country;
       END//
Query OK, 0 rows affected (0.01 sec)

mysql> delimiter ;

mysql> CALL citycount('JPN', @cities); -- cities in Japan
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @cities;
+---------+
| @cities |
+---------+
|     248 |
+---------+
```

```
1 row in set (0.00 sec)

mysql> CALL citycount('FRA', @cities); -- cities in France
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @cities;
+---------+
| @cities |
+---------+
|      40 |
+---------+
1 row in set (0.00 sec)
```

The example uses the `mysql` client `delimiter` command to change the statement delimiter from `;` to `//` while the procedure is being defined. This enables the `;` delimiter used in the procedure body to be passed through to the server rather than being interpreted by `mysql` itself. See Section 25.1, “Defining Stored Programs”.

The `RETURNS` clause may be specified only for a `FUNCTION`, for which it is mandatory. It indicates the return type of the function, and the function body must contain a `RETURN` *`value`* statement. If the `RETURN` statement returns a value of a different type, the value is coerced to the proper type. For example, if a function specifies an `ENUM` or `SET` value in the `RETURNS` clause, but the `RETURN` statement returns an integer, the value returned from the function is the string for the corresponding `ENUM` member of set of `SET` members.

The following example function takes a parameter, performs an operation using an SQL function, and returns the result. In this case, it is unnecessary to use `delimiter` because the function definition contains no internal `;` statement delimiters:

```
mysql> CREATE FUNCTION hello (s CHAR(20))
mysql> RETURNS CHAR(50) DETERMINISTIC
       RETURN CONCAT('Hello, ',s,'!');
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT hello('world');
+----------------+
| hello('world') |
+----------------+
| Hello, world!  |
+----------------+
1 row in set (0.00 sec)
```

Parameter types and function return types can be declared to use any valid data type. The `COLLATE` attribute can be used if preceded by a `CHARACTER SET` specification.

The *`routine_body`* consists of a valid SQL routine statement. This can be a simple statement such as `SELECT` or `INSERT`, or a compound statement written using `BEGIN` and `END`. Compound statements can contain declarations, loops, and other control structure statements. The syntax for these statements is described in Section 13.6, “Compound Statement Syntax”. In practice, stored functions tend to use compound statements, unless the body consists of a single `RETURN` statement.

MySQL permits routines to contain DDL statements, such as `CREATE` and `DROP`. MySQL also permits stored procedures (but not stored functions) to contain SQL transaction statements such as `COMMIT`. Stored functions may not contain statements that perform explicit or implicit commit or rollback. Support for these statements is not required by the SQL standard, which states that each DBMS vendor may decide whether to permit them.

Statements that return a result set can be used within a stored procedure but not within a stored function. This prohibition includes `SELECT` statements that do not have an `INTO` *`var_list`* clause and other statements such as `SHOW`, `EXPLAIN`, and `CHECK TABLE`. For statements that can be determined at function definition time to return a result set, a `Not allowed to return a result set from a function` error occurs (`ER_SP_NO_RETSET`). For statements that can be determined only at runtime to return a result set, a `PROCEDURE %s can't return a result set in the given context` error occurs (`ER_SP_BADSELECT`).

`USE` statements within stored routines are not permitted. When a routine is invoked, an implicit `USE` *`db_name`* is performed (and undone when the routine terminates). The causes the routine to have the given default database while it executes. References to objects in databases other than the routine default database should be qualified with the appropriate database name.

For additional information about statements that are not permitted in stored routines, see Section 25.8, “Restrictions on Stored Programs”.

For information about invoking stored procedures from within programs written in a language that has a MySQL interface, see Section 13.2.1, “CALL Statement”.

MySQL stores the `sql_mode` system variable setting in effect when a routine is created or altered, and always executes the routine with this setting in force, *regardless of the current server SQL mode when the routine begins executing*.

The switch from the SQL mode of the invoker to that of the routine occurs after evaluation of arguments and assignment of the resulting values to routine parameters. If you define a routine in strict SQL mode but invoke it in nonstrict mode, assignment of arguments to routine parameters does not take place in strict mode. If you require that expressions passed to a routine be assigned in strict SQL mode, you should invoke the routine with strict mode in effect.

The `COMMENT` characteristic is a MySQL extension, and may be used to describe the stored routine. This information is displayed by the `SHOW CREATE PROCEDURE` and `SHOW CREATE FUNCTION` statements.

The `LANGUAGE` characteristic indicates the language in which the routine is written. The server ignores this characteristic; only SQL routines are supported.

A routine is considered “deterministic” if it always produces the same result for the same input parameters, and “not deterministic” otherwise. If neither `DETERMINISTIC` nor `NOT DETERMINISTIC` is given in the routine definition, the default is `NOT DETERMINISTIC`. To declare that a function is deterministic, you must specify `DETERMINISTIC` explicitly.

Assessment of the nature of a routine is based on the “honesty” of the creator: MySQL does not check that a routine declared `DETERMINISTIC` is free of statements that produce nondeterministic results. However, misdeclaring a routine might affect results or affect performance. Declaring a nondeterministic routine as `DETERMINISTIC` might lead to unexpected results by causing the optimizer to make incorrect execution plan choices. Declaring a deterministic routine as `NONDETERMINISTIC` might diminish performance by causing available optimizations not to be used.

If binary logging is enabled, the `DETERMINISTIC` characteristic affects which routine definitions MySQL accepts. See Section 25.7, “Stored Program Binary Logging”.

A routine that contains the `NOW()` function (or its synonyms) or `RAND()` is nondeterministic, but it might still be replication-safe. For `NOW()`, the binary log includes the timestamp and replicates correctly. `RAND()` also replicates correctly as long as it is called only a single time during the execution of a routine. (You can consider the routine execution timestamp and random number seed as implicit inputs that are identical on the source and replica.)

Several characteristics provide information about the nature of data use by the routine. In MySQL, these characteristics are advisory only. The server does not use them to constrain what kinds of statements a routine is permitted to execute.

- `CONTAINS SQL` indicates that the routine does not contain statements that read or write data. This is the default if none of these characteristics is given explicitly. Examples of such statements are `SET @x = 1` or `DO RELEASE_LOCK('abc')`, which execute but neither read nor write data.
- `NO SQL` indicates that the routine contains no SQL statements.
- `READS SQL DATA` indicates that the routine contains statements that read data (for example, `SELECT`), but not statements that write data.

- `MODIFIES SQL DATA` indicates that the routine contains statements that may write data (for example, `INSERT` or `DELETE`).

The `SQL SECURITY` characteristic can be `DEFINER` or `INVOKER` to specify the security context; that is, whether the routine executes using the privileges of the account named in the routine `DEFINER` clause or the user who invokes it. This account must have permission to access the database with which the routine is associated. The default value is `DEFINER`. The user who invokes the routine must have the `EXECUTE` privilege for it, as must the `DEFINER` account if the routine executes in definer security context.

The `DEFINER` clause specifies the MySQL account to be used when checking access privileges at routine execution time for routines that have the `SQL SECURITY DEFINER` characteristic.

If the `DEFINER` clause is present, the *`user`* value should be a MySQL account specified as `'user_name'@'host_name'`, `CURRENT_USER`, or `CURRENT_USER()`. The permitted *`user`* values depend on the privileges you hold, as discussed in Section 25.6, “Stored Object Access Control”. Also see that section for additional information about stored routine security.

If the `DEFINER` clause is omitted, the default definer is the user who executes the `CREATE PROCEDURE` or `CREATE FUNCTION` statement. This is the same as specifying `DEFINER = CURRENT_USER` explicitly.

Within the body of a stored routine that is defined with the `SQL SECURITY DEFINER` characteristic, the `CURRENT_USER` function returns the routine's `DEFINER` value. For information about user auditing within stored routines, see Section 6.2.23, “SQL-Based Account Activity Auditing”.

Consider the following procedure, which displays a count of the number of MySQL accounts listed in the `mysql.user` system table:

```
CREATE DEFINER = 'admin'@'localhost' PROCEDURE account_count()
BEGIN
  SELECT 'Number of accounts:', COUNT(*) FROM mysql.user;
END;
```

The procedure is assigned a `DEFINER` account of `'admin'@'localhost'` no matter which user defines it. It executes with the privileges of that account no matter which user invokes it (because the default security characteristic is `DEFINER`). The procedure succeeds or fails depending on whether invoker has the `EXECUTE` privilege for it and `'admin'@'localhost'` has the `SELECT` privilege for the `mysql.user` table.

Now suppose that the procedure is defined with the `SQL SECURITY INVOKER` characteristic:

```
CREATE DEFINER = 'admin'@'localhost' PROCEDURE account_count()
SQL SECURITY INVOKER
BEGIN
  SELECT 'Number of accounts:', COUNT(*) FROM mysql.user;
END;
```

The procedure still has a `DEFINER` of `'admin'@'localhost'`, but in this case, it executes with the privileges of the invoking user. Thus, the procedure succeeds or fails depending on whether the invoker has the `EXECUTE` privilege for it and the `SELECT` privilege for the `mysql.user` table.

By default, when a routine with the `SQL SECURITY DEFINER` characteristic is executed, MySQL Server does not set any active roles for the MySQL account named in the `DEFINER` clause, only the default roles. The exception is if the `activate_all_roles_on_login` system variable is enabled, in which case MySQL Server sets all roles granted to the `DEFINER` user, including mandatory roles. Any privileges granted through roles are therefore not checked by default when the `CREATE PROCEDURE` or `CREATE FUNCTION` statement is issued. For stored programs, if execution should occur with roles different from the default, the program body can execute `SET ROLE` to activate the required roles. This must be done with caution since the privileges assigned to roles can be changed.

The server handles the data type of a routine parameter, local routine variable created with `DECLARE`, or function return value as follows:

- Assignments are checked for data type mismatches and overflow. Conversion and overflow problems result in warnings, or errors in strict SQL mode.
- Only scalar values can be assigned. For example, a statement such as `SET x = (SELECT 1, 2)` is invalid.
- For character data types, if `CHARACTER SET` is included in the declaration, the specified character set and its default collation is used. If the `COLLATE` attribute is also present, that collation is used rather than the default collation.

  If `CHARACTER SET` and `COLLATE` are not present, the database character set and collation in effect at routine creation time are used. To avoid having the server use the database character set and collation, provide an explicit `CHARACTER SET` and a `COLLATE` attribute for character data parameters.

  If you alter the database default character set or collation, stored routines that are to use the new database defaults must be dropped and recreated.

  The database character set and collation are given by the value of the `character_set_database` and `collation_database` system variables. For more information, see Section 10.3.3, “Database Character Set and Collation”.

## 13.1.18 CREATE SERVER Statement

```
CREATE SERVER server_name
    FOREIGN DATA WRAPPER wrapper_name
    OPTIONS (option [, option] ...)

option: {
    HOST character-literal
  | DATABASE character-literal
  | USER character-literal
  | PASSWORD character-literal
  | SOCKET character-literal
  | OWNER character-literal
  | PORT numeric-literal
}
```

This statement creates the definition of a server for use with the `FEDERATED` storage engine. The `CREATE SERVER` statement creates a new row in the `servers` table in the `mysql` database. This statement requires the `SUPER` privilege.

The *`server_name`* should be a unique reference to the server. Server definitions are global within the scope of the server, it is not possible to qualify the server definition to a specific database. *`server_name`* has a maximum length of 64 characters (names longer than 64 characters are silently truncated), and is case-insensitive. You may specify the name as a quoted string.

The *`wrapper_name`* is an identifier and may be quoted with single quotation marks.

For each *`option`* you must specify either a character literal or numeric literal. Character literals are UTF-8, support a maximum length of 64 characters and default to a blank (empty) string. String literals are silently truncated to 64 characters. Numeric literals must be a number between 0 and 9999, default value is 0.

**Note**

The `OWNER` option is currently not applied, and has no effect on the ownership or operation of the server connection that is created.

The `CREATE SERVER` statement creates an entry in the `mysql.servers` table that can later be used with the `CREATE TABLE` statement when creating a `FEDERATED` table. The options that you specify are used to populate the columns in the `mysql.servers` table. The table columns are `Server_name`, `Host`, `Db`, `Username`, `Password`, `Port` and `Socket`.

For example:

```
CREATE SERVER s
FOREIGN DATA WRAPPER mysql
OPTIONS (USER 'Remote', HOST '198.51.100.106', DATABASE 'test');
```

Be sure to specify all options necessary to establish a connection to the server. The user name, host name, and database name are mandatory. Other options might be required as well, such as password.

The data stored in the table can be used when creating a connection to a `FEDERATED` table:

```
CREATE TABLE t (s1 INT) ENGINE=FEDERATED CONNECTION='s';
```

For more information, see Section 16.8, “The FEDERATED Storage Engine”.

`CREATE SERVER` causes an implicit commit. See Section 13.3.3, “Statements That Cause an Implicit Commit”.

`CREATE SERVER` is not written to the binary log, regardless of the logging format that is in use.

## 13.1.19 CREATE SPATIAL REFERENCE SYSTEM Statement

```
CREATE OR REPLACE SPATIAL REFERENCE SYSTEM
    srid srs_attribute ...

CREATE SPATIAL REFERENCE SYSTEM
    [IF NOT EXISTS]
    srid srs_attribute ...

srs_attribute: {
    NAME 'srs_name'
  | DEFINITION 'definition'
  | ORGANIZATION 'org_name' IDENTIFIED BY org_id
  | DESCRIPTION 'description'
}

srid, org_id: 32-bit unsigned integer
```

This statement creates a spatial reference system (SRS) definition and stores it in the data dictionary. It requires the `SUPER` privilege. The resulting data dictionary entry can be inspected using the `INFORMATION_SCHEMA ST_SPATIAL_REFERENCE_SYSTEMS` table.

SRID values must be unique, so if neither `OR REPLACE` nor `IF NOT EXISTS` is specified, an error occurs if an SRS definition with the given *`srid`* value already exists.

With `CREATE OR REPLACE` syntax, any existing SRS definition with the same SRID value is replaced, unless the SRID value is used by some column in an existing table. In that case, an error occurs. For example:

```
mysql> CREATE OR REPLACE SPATIAL REFERENCE SYSTEM 4326 ...;
ERROR 3716 (SR005): Can't modify SRID 4326. There is at
least one column depending on it.
```

To identify which column or columns use the SRID, use this query, replacing 4326 with the SRID of the definition you are trying to create:

```
SELECT * FROM INFORMATION_SCHEMA.ST_GEOMETRY_COLUMNS WHERE SRS_ID=4326;
```

With `CREATE ... IF NOT EXISTS` syntax, any existing SRS definition with the same SRID value causes the new definition to be ignored and a warning occurs.

SRID values must be in the range of 32-bit unsigned integers, with these restrictions:

- SRID 0 is a valid SRID but cannot be used with `CREATE SPATIAL REFERENCE SYSTEM`.
- If the value is in a reserved SRID range, a warning occurs. Reserved ranges are [0, 32767] (reserved by EPSG), [60,000,000, 69,999,999] (reserved by EPSG), and [2,000,000,000, 2,147,483,647] (reserved by MySQL). EPSG stands for the European Petroleum Survey Group.

- Users should not create SRSs with SRIDs in the reserved ranges. Doing so runs the risk of the SRIDs conflicting with future SRS definitions distributed with MySQL, with the result that the new system-provided SRSs are not installed for MySQL upgrades or that the user-defined SRSs are overwritten.

Attributes for the statement must satisfy these conditions:

- Attributes can be given in any order, but no attribute can be given more than once.
- The `NAME` and `DEFINITION` attributes are mandatory.
- The `NAME` *`srs_name`* attribute value must be unique. The combination of the `ORGANIZATION` *`org_name`* and *`org_id`* attribute values must be unique.
- The `NAME` *`srs_name`* attribute value and `ORGANIZATION` *`org_name`* attribute value cannot be empty or begin or end with whitespace.
- String values in attribute specifications cannot contain control characters, including newline.
- The following table shows the maximum lengths for string attribute values.

**Table 13.6 CREATE SPATIAL REFERENCE SYSTEM Attribute Lengths**

| Attribute | Maximum Length (characters) |
|---|---|
| `NAME` | 80 |
| `DEFINITION` | 4096 |
| `ORGANIZATION` | 256 |
| `DESCRIPTION` | 2048 |

Here is an example `CREATE SPATIAL REFERENCE SYSTEM` statement. The `DEFINITION` value is reformatted across multiple lines for readability. (For the statement to be legal, the value actually must be given on a single line.)

```
CREATE SPATIAL REFERENCE SYSTEM 4120
NAME 'Greek'
ORGANIZATION 'EPSG' IDENTIFIED BY 4120
DEFINITION
  'GEOGCS["Greek",DATUM["Greek",SPHEROID["Bessel 1841",
  6377397.155,299.1528128,AUTHORITY["EPSG","7004"]],
  AUTHORITY["EPSG","6120"]],PRIMEM["Greenwich",0,
  AUTHORITY["EPSG","8901"]],UNIT["degree",0.017453292519943278,
  AUTHORITY["EPSG","9122"]],AXIS["Lat",NORTH],AXIS["Lon",EAST],
  AUTHORITY["EPSG","4120"]]';
```

The grammar for SRS definitions is based on the grammar defined in *OpenGIS Implementation Specification: Coordinate Transformation Services*, Revision 1.00, OGC 01-009, January 12, 2001, Section 7.2. This specification is available at http://www.opengeospatial.org/standards/ct.

MySQL incorporates these changes to the specification:

- Only the `<horz cs>` production rule is implemented (that is, geographic and projected SRSs).
- There is an optional, nonstandard `<authority>` clause for `<parameter>`. This makes it possible to recognize projection parameters by authority instead of name.
- The specification does not make `AXIS` clauses mandatory in `GEOGCS` spatial reference system definitions. However, if there are no `AXIS` clauses, MySQL cannot determine whether a definition has axes in latitude-longitude order or longitude-latitude order. MySQL enforces the nonstandard requirement that each `GEOGCS` definition must include two `AXIS` clauses. One must be `NORTH` or `SOUTH`, and the other `EAST` or `WEST`. The `AXIS` clause order determines whether the definition has axes in latitude-longitude order or longitude-latitude order.
- SRS definitions may not contain newlines.

If an SRS definition specifies an authority code for the projection (which is recommended), an error occurs if the definition is missing mandatory parameters. In this case, the error message indicates what the problem is. The projection methods and mandatory parameters that MySQL supports are shown in Table 13.7, “Supported Spatial Reference System Projection Methods” and Table 13.8, “Spatial Reference System Projection Parameters”.

For additional information about writing SRS definitions for MySQL, see Geographic Spatial Reference Systems in MySQL 8.0 and Projected Spatial Reference Systems in MySQL 8.0

The following table shows the projection methods that MySQL supports. MySQL permits unknown projection methods but cannot check the definition for mandatory parameters and cannot convert spatial data to or from an unknown projection. For detailed explanations of how each projection works, including formulas, see EPSG Guidance Note 7-2.

**Table 13.7 Supported Spatial Reference System Projection Methods**

| EPSG Code | Projection Name | Mandatory Parameters (EPSG Codes) |
|---|---|---|
| 1024 | Popular Visualisation Pseudo Mercator | 8801, 8802, 8806, 8807 |
| 1027 | Lambert Azimuthal Equal Area (Spherical) | 8801, 8802, 8806, 8807 |
| 1028 | Equidistant Cylindrical | 8823, 8802, 8806, 8807 |
| 1029 | Equidistant Cylindrical (Spherical) | 8823, 8802, 8806, 8807 |
| 1041 | Krovak (North Orientated) | 8811, 8833, 1036, 8818, 8819, 8806, 8807 |
| 1042 | Krovak Modified | 8811, 8833, 1036, 8818, 8819, 8806, 8807, 8617, 8618, 1026, 1027, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035 |
| 1043 | Krovak Modified (North Orientated) | 8811, 8833, 1036, 8818, 8819, 8806, 8807, 8617, 8618, 1026, 1027, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035 |
| 1051 | Lambert Conic Conformal (2SP Michigan) | 8821, 8822, 8823, 8824, 8826, 8827, 1038 |
| 1052 | Colombia Urban | 8801, 8802, 8806, 8807, 1039 |
| 9801 | Lambert Conic Conformal (1SP) | 8801, 8802, 8805, 8806, 8807 |
| 9802 | Lambert Conic Conformal (2SP) | 8821, 8822, 8823, 8824, 8826, 8827 |
| 9803 | Lambert Conic Conformal (2SP Belgium) | 8821, 8822, 8823, 8824, 8826, 8827 |
| 9804 | Mercator (variant A) | 8801, 8802, 8805, 8806, 8807 |
| 9805 | Mercator (variant B) | 8823, 8802, 8806, 8807 |
| 9806 | Cassini-Soldner | 8801, 8802, 8806, 8807 |
| 9807 | Transverse Mercator | 8801, 8802, 8805, 8806, 8807 |
| 9808 | Transverse Mercator (South Orientated) | 8801, 8802, 8805, 8806, 8807 |
| 9809 | Oblique Stereographic | 8801, 8802, 8805, 8806, 8807 |
| 9810 | Polar Stereographic (variant A) | 8801, 8802, 8805, 8806, 8807 |
| 9811 | New Zealand Map Grid | 8801, 8802, 8806, 8807 |

| EPSG Code | Projection Name | Mandatory Parameters (EPSG Codes) |
|---|---|---|
| 9812 | Hotine Oblique Mercator (variant A) | 8811, 8812, 8813, 8814, 8815, 8806, 8807 |
| 9813 | Laborde Oblique Mercator | 8811, 8812, 8813, 8815, 8806, 8807 |
| 9815 | Hotine Oblique Mercator (variant B) | 8811, 8812, 8813, 8814, 8815, 8816, 8817 |
| 9816 | Tunisia Mining Grid | 8821, 8822, 8826, 8827 |
| 9817 | Lambert Conic Near-Conformal | 8801, 8802, 8805, 8806, 8807 |
| 9818 | American Polyconic | 8801, 8802, 8806, 8807 |
| 9819 | Krovak | 8811, 8833, 1036, 8818, 8819, 8806, 8807 |
| 9820 | Lambert Azimuthal Equal Area | 8801, 8802, 8806, 8807 |
| 9822 | Albers Equal Area | 8821, 8822, 8823, 8824, 8826, 8827 |
| 9824 | Transverse Mercator Zoned Grid System | 8801, 8830, 8831, 8805, 8806, 8807 |
| 9826 | Lambert Conic Conformal (West Orientated) | 8801, 8802, 8805, 8806, 8807 |
| 9828 | Bonne (South Orientated) | 8801, 8802, 8806, 8807 |
| 9829 | Polar Stereographic (variant B) | 8832, 8833, 8806, 8807 |
| 9830 | Polar Stereographic (variant C) | 8832, 8833, 8826, 8827 |
| 9831 | Guam Projection | 8801, 8802, 8806, 8807 |
| 9832 | Modified Azimuthal Equidistant | 8801, 8802, 8806, 8807 |
| 9833 | Hyperbolic Cassini-Soldner | 8801, 8802, 8806, 8807 |
| 9834 | Lambert Cylindrical Equal Area (Spherical) | 8823, 8802, 8806, 8807 |
| 9835 | Lambert Cylindrical Equal Area | 8823, 8802, 8806, 8807 |

The following table shows the projection parameters that MySQL recognizes. Recognition occurs primarily by authority code. If there is no authority code, MySQL falls back to case-insensitive string matching on the parameter name. For details about each parameter, look it up by code in the EPSG Online Registry.

**Table 13.8 Spatial Reference System Projection Parameters**

| EPSG Code | Fallback Name (Recognized by MySQL) | EPSG Name |
|---|---|---|
| 1026 | c1 | C1 |
| 1027 | c2 | C2 |
| 1028 | c3 | C3 |
| 1029 | c4 | C4 |
| 1030 | c5 | C5 |
| 1031 | c6 | C6 |
| 1032 | c7 | C7 |
| 1033 | c8 | C8 |
| 1034 | c9 | C9 |

| EPSG Code | Fallback Name (Recognized by MySQL) | EPSG Name |
|---|---|---|
| 1035 | c10 | C10 |
| 1036 | azimuth | Co-latitude of cone axis |
| 1038 | ellipsoid_scale_factor | Ellipsoid scaling factor |
| 1039 | projection_plane_height_at_origin | Projection plane origin height |
| 8617 | evaluation_point_ordinate_1 | Ordinate 1 of evaluation point |
| 8618 | evaluation_point_ordinate_2 | Ordinate 2 of evaluation point |
| 8801 | latitude_of_origin | Latitude of natural origin |
| 8802 | central_meridian | Longitude of natural origin |
| 8805 | scale_factor | Scale factor at natural origin |
| 8806 | false_easting | False easting |
| 8807 | false_northing | False northing |
| 8811 | latitude_of_center | Latitude of projection centre |
| 8812 | longitude_of_center | Longitude of projection centre |
| 8813 | azimuth | Azimuth of initial line |
| 8814 | rectified_grid_angle | Angle from Rectified to Skew Grid |
| 8815 | scale_factor | Scale factor on initial line |
| 8816 | false_easting | Easting at projection centre |
| 8817 | false_northing | Northing at projection centre |
| 8818 | pseudo_standard_parallel_1 | Latitude of pseudo standard parallel |
| 8819 | scale_factor | Scale factor on pseudo standard parallel |
| 8821 | latitude_of_origin | Latitude of false origin |
| 8822 | central_meridian | Longitude of false origin |
| 8823 | standard_parallel_1, standard_parallel1 | Latitude of 1st standard parallel |
| 8824 | standard_parallel_2, standard_parallel2 | Latitude of 2nd standard parallel |
| 8826 | false_easting | Easting at false origin |
| 8827 | false_northing | Northing at false origin |
| 8830 | initial_longitude | Initial longitude |
| 8831 | zone_width | Zone width |
| 8832 | standard_parallel | Latitude of standard parallel |
| 8833 | longitude_of_center | Longitude of origin |

## 13.1.20 CREATE TABLE Statement

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
    [table_options]
    [partition_options]

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [(create_definition,...)]
    [table_options]
```

```
    [partition_options]
    [IGNORE | REPLACE]
    [AS] query_expression

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    { LIKE old_tbl_name | (LIKE old_tbl_name) }

create_definition: {
    col_name column_definition
  | {INDEX | KEY} [index_name] [index_type] (key_part,...)
      [index_option] ...
  | {FULLTEXT | SPATIAL} [INDEX | KEY] [index_name] (key_part,...)
      [index_option] ...
  | [CONSTRAINT [symbol]] PRIMARY KEY
      [index_type] (key_part,...)
      [index_option] ...
  | [CONSTRAINT [symbol]] UNIQUE [INDEX | KEY]
      [index_name] [index_type] (key_part,...)
      [index_option] ...
  | [CONSTRAINT [symbol]] FOREIGN KEY
      [index_name] (col_name,...)
      reference_definition
  | check_constraint_definition
}

column_definition: {
    data_type [NOT NULL | NULL] [DEFAULT {literal | (expr)} ]
      [VISIBLE | INVISIBLE]
      [AUTO_INCREMENT] [UNIQUE [KEY]] [[PRIMARY] KEY]
      [COMMENT 'string']
      [COLLATE collation_name]
      [COLUMN_FORMAT {FIXED | DYNAMIC | DEFAULT}]
      [ENGINE_ATTRIBUTE [=] 'string']
      [SECONDARY_ENGINE_ATTRIBUTE [=] 'string']
      [STORAGE {DISK | MEMORY}]
      [reference_definition]
      [check_constraint_definition]
  | data_type
      [COLLATE collation_name]
      [GENERATED ALWAYS] AS (expr)
      [VIRTUAL | STORED] [NOT NULL | NULL]
      [VISIBLE | INVISIBLE]
      [UNIQUE [KEY]] [[PRIMARY] KEY]
      [COMMENT 'string']
      [reference_definition]
      [check_constraint_definition]
}

data_type:
    (see Chapter 11, Data Types)

key_part: {col_name [(length)] | (expr)} [ASC | DESC]

index_type:
    USING {BTREE | HASH}

index_option: {
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
  | {VISIBLE | INVISIBLE}
  |ENGINE_ATTRIBUTE [=] 'string'
  |SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
}

check_constraint_definition:
    [CONSTRAINT [symbol]] CHECK (expr) [[NOT] ENFORCED]

reference_definition:
    REFERENCES tbl_name (key_part,...)
      [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
```

```
      [ON DELETE reference_option]
      [ON UPDATE reference_option]

reference_option:
    RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT

table_options:
    table_option [[,] table_option] ...

table_option: {
    AUTOEXTEND_SIZE [=] value
  | AUTO_INCREMENT [=] value
  | AVG_ROW_LENGTH [=] value
  | [DEFAULT] CHARACTER SET [=] charset_name
  | CHECKSUM [=] {0 | 1}
  | [DEFAULT] COLLATE [=] collation_name
  | COMMENT [=] 'string'
  | COMPRESSION [=] {'ZLIB' | 'LZ4' | 'NONE'}
  | CONNECTION [=] 'connect_string'
  | {DATA | INDEX} DIRECTORY [=] 'absolute path to directory'
  | DELAY_KEY_WRITE [=] {0 | 1}
  | ENCRYPTION [=] {'Y' | 'N'}
  | ENGINE [=] engine_name
  | ENGINE_ATTRIBUTE [=] 'string'
  | INSERT_METHOD [=] { NO | FIRST | LAST }
  | KEY_BLOCK_SIZE [=] value
  | MAX_ROWS [=] value
  | MIN_ROWS [=] value
  | PACK_KEYS [=] {0 | 1 | DEFAULT}
  | PASSWORD [=] 'string'
  | ROW_FORMAT [=] {DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUNDANT | COMPACT}
  | START TRANSACTION
  | SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
  | STATS_AUTO_RECALC [=] {DEFAULT | 0 | 1}
  | STATS_PERSISTENT [=] {DEFAULT | 0 | 1}
  | STATS_SAMPLE_PAGES [=] value
  | tablespace_option
  | UNION [=] (tbl_name[,tbl_name]...)
}

partition_options:
    PARTITION BY
        { [LINEAR] HASH(expr)
        | [LINEAR] KEY [ALGORITHM={1 | 2}] (column_list)
        | RANGE{(expr) | COLUMNS(column_list)}
        | LIST{(expr) | COLUMNS(column_list)} }
    [PARTITIONS num]
    [SUBPARTITION BY
        { [LINEAR] HASH(expr)
        | [LINEAR] KEY [ALGORITHM={1 | 2}] (column_list) }
      [SUBPARTITIONS num]
    ]
    [(partition_definition [, partition_definition] ...)]

partition_definition:
    PARTITION partition_name
        [VALUES
            {LESS THAN {(expr | value_list) | MAXVALUE}
            |
            IN (value_list)}]
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'string' ]
        [DATA DIRECTORY [=] 'data_dir']
        [INDEX DIRECTORY [=] 'index_dir']
        [MAX_ROWS [=] max_number_of_rows]
        [MIN_ROWS [=] min_number_of_rows]
        [TABLESPACE [=] tablespace_name]
        [(subpartition_definition [, subpartition_definition] ...)]

subpartition_definition:
    SUBPARTITION logical_name
        [[STORAGE] ENGINE [=] engine_name]
```

```
        [COMMENT [=] 'string' ]
        [DATA DIRECTORY [=] 'data_dir']
        [INDEX DIRECTORY [=] 'index_dir']
        [MAX_ROWS [=] max_number_of_rows]
        [MIN_ROWS [=] min_number_of_rows]
        [TABLESPACE [=] tablespace_name]

tablespace_option:
    TABLESPACE tablespace_name [STORAGE DISK]
  | [TABLESPACE tablespace_name] STORAGE MEMORY

query_expression:
    SELECT ...   (Some valid select or union statement)
```

`CREATE TABLE` creates a table with the given name. You must have the `CREATE` privilege for the table.

By default, tables are created in the default database, using the `InnoDB` storage engine. An error occurs if the table exists, if there is no default database, or if the database does not exist.

MySQL has no limit on the number of tables. The underlying file system may have a limit on the number of files that represent tables. Individual storage engines may impose engine-specific constraints. `InnoDB` permits up to 4 billion tables.

For information about the physical representation of a table, see Section 13.1.20.1, “Files Created by CREATE TABLE”.

There are several aspects to the `CREATE TABLE` statement, described under the following topics in this section:

- Table Name
- Temporary Tables
- Table Cloning and Copying
- Column Data Types and Attributes
- Indexes, Foreign Keys, and CHECK Constraints
- Table Options
- Table Partitioning

## Table Name

- *`tbl_name`*

  The table name can be specified as *`db_name.tbl_name`* to create the table in a specific database. This works regardless of whether there is a default database, assuming that the database exists. If you use quoted identifiers, quote the database and table names separately. For example, write `` `mydb`.`mytbl` ``, not `` `mydb.mytbl` ``.

  Rules for permissible table names are given in Section 9.2, “Schema Object Names”.

- `IF NOT EXISTS`

  Prevents an error from occurring if the table exists. However, there is no verification that the existing table has a structure identical to that indicated by the `CREATE TABLE` statement.

## Temporary Tables

You can use the `TEMPORARY` keyword when creating a table. A `TEMPORARY` table is visible only within the current session, and is dropped automatically when the session is closed. For more information, see Section 13.1.20.2, “CREATE TEMPORARY TABLE Statement”.

## Table Cloning and Copying

- `LIKE`

  Use `CREATE TABLE ... LIKE` to create an empty table based on the definition of another table, including any column attributes and indexes defined in the original table:

  ```
  CREATE TABLE new_tbl LIKE orig_tbl;
  ```

  For more information, see Section 13.1.20.3, “CREATE TABLE ... LIKE Statement”.

- `[AS] query_expression`

  To create one table from another, add a `SELECT` statement at the end of the `CREATE TABLE` statement:

  ```
  CREATE TABLE new_tbl AS SELECT * FROM orig_tbl;
  ```

  For more information, see Section 13.1.20.4, “CREATE TABLE ... SELECT Statement”.

- `IGNORE | REPLACE`

  The `IGNORE` and `REPLACE` options indicate how to handle rows that duplicate unique key values when copying a table using a `SELECT` statement.

  For more information, see Section 13.1.20.4, “CREATE TABLE ... SELECT Statement”.

## Column Data Types and Attributes

There is a hard limit of 4096 columns per table, but the effective maximum may be less for a given table and depends on the factors discussed in Section 8.4.7, “Limits on Table Column Count and Row Size”.

- *`data_type`*

  *`data_type`* represents the data type in a column definition. For a full description of the syntax available for specifying column data types, as well as information about the properties of each type, see Chapter 11, *Data Types*.

  - Some attributes do not apply to all data types. `AUTO_INCREMENT` applies only to integer and floating-point types. Prior to MySQL 8.0.13, `DEFAULT` does not apply to the `BLOB`, `TEXT`, `GEOMETRY`, and `JSON` types.

  - Character data types (`CHAR`, `VARCHAR`, the `TEXT` types, `ENUM`, `SET`, and any synonyms) can include `CHARACTER SET` to specify the character set for the column. `CHARSET` is a synonym for `CHARACTER SET`. A collation for the character set can be specified with the `COLLATE` attribute, along with any other attributes. For details, see Chapter 10, *Character Sets, Collations, Unicode*. Example:

    ```
    CREATE TABLE t (c CHAR(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin);
    ```

    MySQL 8.0 interprets length specifications in character column definitions in characters. Lengths for `BINARY` and `VARBINARY` are in bytes.

  - For `CHAR`, `VARCHAR`, `BINARY`, and `VARBINARY` columns, indexes can be created that use only the leading part of column values, using *`col_name(length)`* syntax to specify an index prefix length. `BLOB` and `TEXT` columns also can be indexed, but a prefix length *must* be given. Prefix lengths are given in characters for nonbinary string types and in bytes for binary string types. That is, index entries consist of the first *`length`* characters of each column value for `CHAR`, `VARCHAR`, and `TEXT` columns, and the first *`length`* bytes of each column value for `BINARY`, `VARBINARY`, and `BLOB` columns. Indexing only a prefix of column values like this can make the index file much smaller. For additional information about index prefixes, see Section 13.1.15, “CREATE INDEX Statement”.

Only the InnoDB and MyISAM storage engines support indexing on BLOB and TEXT columns. For example:

```
CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));
```

If a specified index prefix exceeds the maximum column data type size, CREATE TABLE handles the index as follows:

- For a nonunique index, either an error occurs (if strict SQL mode is enabled), or the index length is reduced to lie within the maximum column data type size and a warning is produced (if strict SQL mode is not enabled).
- For a unique index, an error occurs regardless of SQL mode because reducing the index length might enable insertion of nonunique entries that do not meet the specified uniqueness requirement.

- JSON columns cannot be indexed. You can work around this restriction by creating an index on a generated column that extracts a scalar value from the JSON column. See Indexing a Generated Column to Provide a JSON Column Index, for a detailed example.

- NOT NULL | NULL

  If neither NULL nor NOT NULL is specified, the column is treated as though NULL had been specified.

  In MySQL 8.0, only the InnoDB, MyISAM, and MEMORY storage engines support indexes on columns that can have NULL values. In other cases, you must declare indexed columns as NOT NULL or an error results.

- DEFAULT

  Specifies a default value for a column. For more information about default value handling, including the case that a column definition includes no explicit DEFAULT value, see Section 11.6, “Data Type Default Values”.

  If the NO_ZERO_DATE or NO_ZERO_IN_DATE SQL mode is enabled and a date-valued default is not correct according to that mode, CREATE TABLE produces a warning if strict SQL mode is not enabled and an error if strict mode is enabled. For example, with NO_ZERO_IN_DATE enabled, c1 DATE DEFAULT '2010-00-00' produces a warning.

- VISIBLE, INVISIBLE

  Specify column visibility. The default is VISIBLE if neither keyword is present. A table must have at least one visible column. Attempting to make all columns invisible produces an error. For more information, see Section 13.1.20.10, “Invisible Columns”.

  The VISIBLE and INVISIBLE keywords are available as of MySQL 8.0.23. Prior to MySQL 8.0.23, all columns are visible.

- AUTO_INCREMENT

  An integer or floating-point column can have the additional attribute AUTO_INCREMENT. When you insert a value of NULL (recommended) or 0 into an indexed AUTO_INCREMENT column, the column

is set to the next sequence value. Typically this is *value*+1, where *value* is the largest value for the column currently in the table. AUTO_INCREMENT sequences begin with 1.

To retrieve an AUTO_INCREMENT value after inserting a row, use the LAST_INSERT_ID() SQL function or the mysql_insert_id() C API function. See Section 12.16, “Information Functions”, and mysql_insert_id().

If the NO_AUTO_VALUE_ON_ZERO SQL mode is enabled, you can store 0 in AUTO_INCREMENT columns as 0 without generating a new sequence value. See Section 5.1.11, “Server SQL Modes”.

There can be only one AUTO_INCREMENT column per table, it must be indexed, and it cannot have a DEFAULT value. An AUTO_INCREMENT column works properly only if it contains only positive values. Inserting a negative number is regarded as inserting a very large positive number. This is done to avoid precision problems when numbers “wrap” over from positive to negative and also to ensure that you do not accidentally get an AUTO_INCREMENT column that contains 0.

For MyISAM tables, you can specify an AUTO_INCREMENT secondary column in a multiple-column key. See Section 3.6.9, “Using AUTO_INCREMENT”.

To make MySQL compatible with some ODBC applications, you can find the AUTO_INCREMENT value for the last inserted row with the following query:

```
SELECT * FROM tbl_name WHERE auto_col IS NULL
```

This method requires that sql_auto_is_null variable is not set to 0. See Section 5.1.8, “Server System Variables”.

For information about InnoDB and AUTO_INCREMENT, see Section 15.6.1.6, “AUTO_INCREMENT Handling in InnoDB”. For information about AUTO_INCREMENT and MySQL Replication, see Section 17.5.1.1, “Replication and AUTO_INCREMENT”.

- COMMENT

  A comment for a column can be specified with the COMMENT option, up to 1024 characters long. The comment is displayed by the SHOW CREATE TABLE and SHOW FULL COLUMNS statements. It is also shown in the COLUMN_COMMENT column of the Information Schema COLUMNS table.

- COLUMN_FORMAT

  In NDB Cluster, it is also possible to specify a data storage format for individual columns of NDB tables using COLUMN_FORMAT. Permissible column formats are FIXED, DYNAMIC, and DEFAULT. FIXED is used to specify fixed-width storage, DYNAMIC permits the column to be variable-width, and DEFAULT causes the column to use fixed-width or variable-width storage as determined by the column's data type (possibly overridden by a ROW_FORMAT specifier).

  For NDB tables, the default value for COLUMN_FORMAT is FIXED.

  In NDB Cluster, the maximum possible offset for a column defined with COLUMN_FORMAT=FIXED is 8188 bytes. For more information and possible workarounds, see Section 23.2.7.5, “Limits Associated with Database Objects in NDB Cluster”.

  COLUMN_FORMAT currently has no effect on columns of tables using storage engines other than NDB. MySQL 8.0 silently ignores COLUMN_FORMAT.

- ENGINE_ATTRIBUTE and SECONDARY_ENGINE_ATTRIBUTE options (available as of MySQL 8.0.21) are used to specify column attributes for primary and secondary storage engines. The options are reserved for future use.

  Permitted values are a string literal containing a valid JSON document or an empty string (''). Invalid JSON is rejected.

  ```
  CREATE TABLE t1 (c1 INT ENGINE_ATTRIBUTE='{"key":"value"}');
  ```

ENGINE_ATTRIBUTE and SECONDARY_ENGINE_ATTRIBUTE values can be repeated without error. In this case, the last specified value is used.

ENGINE_ATTRIBUTE and SECONDARY_ENGINE_ATTRIBUTE values are not checked by the server, nor are they cleared when the table's storage engine is changed.

- STORAGE

  For NDB tables, it is possible to specify whether the column is stored on disk or in memory by using a STORAGE clause. STORAGE DISK causes the column to be stored on disk, and STORAGE MEMORY causes in-memory storage to be used. The CREATE TABLE statement used must still include a TABLESPACE clause:

```
mysql> CREATE TABLE t1 (
    ->     c1 INT STORAGE DISK,
    ->     c2 INT STORAGE MEMORY
    -> ) ENGINE NDB;
ERROR 1005 (HY000): Can't create table 'c.t1' (errno: 140)

mysql> CREATE TABLE t1 (
    ->     c1 INT STORAGE DISK,
    ->     c2 INT STORAGE MEMORY
    -> ) TABLESPACE ts_1 ENGINE NDB;
Query OK, 0 rows affected (1.06 sec)
```

  For NDB tables, STORAGE DEFAULT is equivalent to STORAGE MEMORY.

  The STORAGE clause has no effect on tables using storage engines other than NDB. The STORAGE keyword is supported only in the build of mysqld that is supplied with NDB Cluster; it is not recognized in any other version of MySQL, where any attempt to use the STORAGE keyword causes a syntax error.

- GENERATED ALWAYS

  Used to specify a generated column expression. For information about generated columns, see Section 13.1.20.8, “CREATE TABLE and Generated Columns”.

  Stored generated columns can be indexed. InnoDB supports secondary indexes on virtual generated columns. See Section 13.1.20.9, “Secondary Indexes and Generated Columns”.

## Indexes, Foreign Keys, and CHECK Constraints

Several keywords apply to creation of indexes, foreign keys, and CHECK constraints. For general background in addition to the following descriptions, see Section 13.1.15, “CREATE INDEX Statement”, Section 13.1.20.5, “FOREIGN KEY Constraints”, and Section 13.1.20.6, “CHECK Constraints”.

- CONSTRAINT *symbol*

  The CONSTRAINT *symbol* clause may be given to name a constraint. If the clause is not given, or a *symbol* is not included following the CONSTRAINT keyword, MySQL automatically generates a constraint name, with the exception noted below. The *symbol* value, if used, must be unique per schema (database), per constraint type. A duplicate *symbol* results in an error. See also the discussion about length limits of generated constraint identifiers at Section 9.2.1, “Identifier Length Limits”.

**Note**

If the CONSTRAINT *symbol* clause is not given in a foreign key definition, or a *symbol* is not included following the CONSTRAINT keyword, MySQL uses the foreign key index name up to MySQL 8.0.15, and automatically generates a constraint name thereafter.

The SQL standard specifies that all types of constraints (primary key, unique index, foreign key, check) belong to the same namespace. In MySQL, each constraint type has its own namespace per schema. Consequently, names for each type of constraint must be unique per schema, but constraints of different types can have the same name.

- `PRIMARY KEY`

  A unique index where all key columns must be defined as `NOT NULL`. If they are not explicitly declared as `NOT NULL`, MySQL declares them so implicitly (and silently). A table can have only one `PRIMARY KEY`. The name of a `PRIMARY KEY` is always `PRIMARY`, which thus cannot be used as the name for any other kind of index.

  If you do not have a `PRIMARY KEY` and an application asks for the `PRIMARY KEY` in your tables, MySQL returns the first `UNIQUE` index that has no `NULL` columns as the `PRIMARY KEY`.

  In `InnoDB` tables, keep the `PRIMARY KEY` short to minimize storage overhead for secondary indexes. Each secondary index entry contains a copy of the primary key columns for the corresponding row. (See Section 15.6.2.1, “Clustered and Secondary Indexes”.)

  In the created table, a `PRIMARY KEY` is placed first, followed by all `UNIQUE` indexes, and then the nonunique indexes. This helps the MySQL optimizer to prioritize which index to use and also more quickly to detect duplicated `UNIQUE` keys.

  A `PRIMARY KEY` can be a multiple-column index. However, you cannot create a multiple-column index using the `PRIMARY KEY` key attribute in a column specification. Doing so only marks that single column as primary. You must use a separate `PRIMARY KEY(key_part, ...)` clause.

  If a table has a `PRIMARY KEY` or `UNIQUE NOT NULL` index that consists of a single column that has an integer type, you can use `_rowid` to refer to the indexed column in `SELECT` statements, as described in Unique Indexes.

  In MySQL, the name of a `PRIMARY KEY` is `PRIMARY`. For other indexes, if you do not assign a name, the index is assigned the same name as the first indexed column, with an optional suffix (`_2`, `_3`, `...`) to make it unique. You can see index names for a table using `SHOW INDEX FROM tbl_name`. See Section 13.7.7.22, “SHOW INDEX Statement”.

- `KEY | INDEX`

  `KEY` is normally a synonym for `INDEX`. The key attribute `PRIMARY KEY` can also be specified as just `KEY` when given in a column definition. This was implemented for compatibility with other database systems.

- `UNIQUE`

  A `UNIQUE` index creates a constraint such that all values in the index must be distinct. An error occurs if you try to add a new row with a key value that matches an existing row. For all engines, a `UNIQUE` index permits multiple `NULL` values for columns that can contain `NULL`. If you specify a prefix value for a column in a `UNIQUE` index, the column values must be unique within the prefix length.

  If a table has a `PRIMARY KEY` or `UNIQUE NOT NULL` index that consists of a single column that has an integer type, you can use `_rowid` to refer to the indexed column in `SELECT` statements, as described in Unique Indexes.

- `FULLTEXT`

  A `FULLTEXT` index is a special type of index used for full-text searches. Only the `InnoDB` and `MyISAM` storage engines support `FULLTEXT` indexes. They can be created only from `CHAR`, `VARCHAR`, and `TEXT` columns. Indexing always happens over the entire column; column prefix indexing is not supported and any prefix length is ignored if specified. See Section 12.10, “Full-

Text Search Functions", for details of operation. A `WITH PARSER` clause can be specified as an *`index_option`* value to associate a parser plugin with the index if full-text indexing and searching operations need special handling. This clause is valid only for `FULLTEXT` indexes. `InnoDB` and `MyISAM` support full-text parser plugins. See Full-Text Parser Plugins and Writing Full-Text Parser Plugins for more information.

- `SPATIAL`

  You can create `SPATIAL` indexes on spatial data types. Spatial types are supported only for `InnoDB` and `MyISAM` tables, and indexed columns must be declared as `NOT NULL`. See Section 11.4, "Spatial Data Types".

- `FOREIGN KEY`

  MySQL supports foreign keys, which let you cross-reference related data across tables, and foreign key constraints, which help keep this spread-out data consistent. For definition and option information, see *`reference_definition`*, and *`reference_option`*.

  Partitioned tables employing the `InnoDB` storage engine do not support foreign keys. See Section 24.6, "Restrictions and Limitations on Partitioning", for more information.

- `CHECK`

  The `CHECK` clause enables the creation of constraints to be checked for data values in table rows. See Section 13.1.20.6, "CHECK Constraints".

- *`key_part`*

  - A *`key_part`* specification can end with `ASC` or `DESC` to specify whether index values are stored in ascending or descending order. The default is ascending if no order specifier is given.

  - Prefixes, defined by the *`length`* attribute, can be up to 767 bytes long for `InnoDB` tables that use the `REDUNDANT` or `COMPACT` row format. The prefix length limit is 3072 bytes for `InnoDB` tables that use the `DYNAMIC` or `COMPRESSED` row format. For `MyISAM` tables, the prefix length limit is 1000 bytes.

    Prefix *limits* are measured in bytes. However, prefix *lengths* for index specifications in `CREATE TABLE`, `ALTER TABLE`, and `CREATE INDEX` statements are interpreted as number of characters for nonbinary string types (`CHAR`, `VARCHAR`, `TEXT`) and number of bytes for binary string types (`BINARY`, `VARBINARY`, `BLOB`). Take this into account when specifying a prefix length for a nonbinary string column that uses a multibyte character set.

  - Beginning with MySQL 8.0.17, the *`expr`* for a *`key_part`* specification can take the form `(CAST` *`json_path`* `AS` *`type`* `ARRAY)` to create a multi-valued index on a `JSON` column. Multi-Valued Indexes, provides detailed information regarding creation of, usage of, and restrictions and limitations on multi-valued indexes.

- *`index_type`*

  Some storage engines permit you to specify an index type when creating an index. The syntax for the *`index_type`* specifier is `USING` *`type_name`*.

  Example:

```
CREATE TABLE lookup
  (id INT, INDEX USING BTREE (id))
  ENGINE = MEMORY;
```

  The preferred position for `USING` is after the index column list. It can be given before the column list, but support for use of the option in that position is deprecated and you should expect it to be removed in a future MySQL release.

- *index_option*

  *index_option* values specify additional options for an index.

  - KEY_BLOCK_SIZE

    For MyISAM tables, KEY_BLOCK_SIZE optionally specifies the size in bytes to use for index key blocks. The value is treated as a hint; a different size could be used if necessary. A KEY_BLOCK_SIZE value specified for an individual index definition overrides the table-level KEY_BLOCK_SIZE value.

    For information about the table-level KEY_BLOCK_SIZE attribute, see Table Options.

  - WITH PARSER

    The WITH PARSER option can be used only with FULLTEXT indexes. It associates a parser plugin with the index if full-text indexing and searching operations need special handling. InnoDB and MyISAM support full-text parser plugins. If you have a MyISAM table with an associated full-text parser plugin, you can convert the table to InnoDB using ALTER TABLE.

  - COMMENT

    Index definitions can include an optional comment of up to 1024 characters.

    You can set the InnoDB MERGE_THRESHOLD value for an individual index using the *index_option* COMMENT clause. See Section 15.8.11, “Configuring the Merge Threshold for Index Pages”.

  - VISIBLE, INVISIBLE

    Specify index visibility. Indexes are visible by default. An invisible index is not used by the optimizer. Specification of index visibility applies to indexes other than primary keys (either explicit or implicit). For more information, see Section 8.3.12, “Invisible Indexes”.

  - ENGINE_ATTRIBUTE and SECONDARY_ENGINE_ATTRIBUTE options (available as of MySQL 8.0.21) are used to specify index attributes for primary and secondary storage engines. The options are reserved for future use.

  For more information about permissible *index_option* values, see Section 13.1.15, “CREATE INDEX Statement”. For more information about indexes, see Section 8.3.1, “How MySQL Uses Indexes”.

- *reference_definition*

  For *reference_definition* syntax details and examples, see Section 13.1.20.5, “FOREIGN KEY Constraints”.

  InnoDB and NDB tables support checking of foreign key constraints. The columns of the referenced table must always be explicitly named. Both ON DELETE and ON UPDATE actions on foreign keys are supported. For more detailed information and examples, see Section 13.1.20.5, “FOREIGN KEY Constraints”.

  For other storage engines, MySQL Server parses and ignores the FOREIGN KEY syntax in CREATE TABLE statements.

  

  **Important**

  For users familiar with the ANSI/ISO SQL Standard, please note that no storage engine, including InnoDB, recognizes or enforces the MATCH clause used in referential integrity constraint definitions. Use of an explicit MATCH clause does not have the specified effect, and also causes ON DELETE and