mysql> SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME = 'e';

+----------------+------------+

| PARTITION\_NAME | TABLE\_ROWS |

+----------------+------------+

| p0

| p1

|

|

0

1000000

|

|

+----------------+------------+

2 rows in set (0.00 sec)

# Exchange partition p0 of table e with the table e2 'WITH VALIDATION'

mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2 WITH VALIDATION;

Query OK, 0 rows affected (0.74 sec)

# Confirm that the partition was exchanged with table e2

mysql> SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME = 'e';

+----------------+------------+

| PARTITION\_NAME | TABLE\_ROWS |

+----------------+------------+

| p0

| p1

|

|

1000000

1000000

|

|

+----------------+------------+

2 rows in set (0.00 sec)

# Once again, drop the rows from p0 of table e

mysql> DELETE FROM e WHERE id < 1000001;

Query OK, 1000000 rows affected (5.55 sec)

# Confirm that there are no rows in partition p0

mysql> SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME = 'e';

+----------------+------------+

| PARTITION\_NAME | TABLE\_ROWS |

+----------------+------------+

| p0

| p1

|

|

0

1000000

|

|

+----------------+------------+

2 rows in set (0.00 sec)

# Exchange partition p0 of table e with the table e3 'WITHOUT VALIDATION'

mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e3 WITHOUT VALIDATION;

Query OK, 0 rows affected (0.01 sec)

# Confirm that the partition was exchanged with table e3

mysql> SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME = 'e';

+----------------+------------+

| PARTITION\_NAME | TABLE\_ROWS |

+----------------+------------+

| p0

| p1

|

|

1000000

1000000

|

|

+----------------+------------+

2 rows in set (0.00 sec)

If a partition is exchanged with a table that contains rows that do not match the partition definition, it is the responsibility of the database administrator to fix the non-matching rows, which can be performed using REPAIR TABLE or ALTER TABLE ... REPAIR PARTITION.

**Exchanging** **a** **Subpartition** **with** **a** **Nonpartitioned** **Table**

You can also exchange a subpartition of a subpartitioned table (see Section 24.2.6, “Subpartitioning”) with a nonpartitioned table using an ALTER TABLE ... EXCHANGE PARTITION statement. In the following example, we first create a table es that is partitioned by RANGE and subpartitioned by KEY, populate this table as we did table e, and then create an empty, nonpartitioned copy es2 of the table, as shown here:

mysql> **CREATE** **TABLE** **es** **(**

-> **id** **INT** **NOT** **NULL,**

-> **fname** **VARCHAR(30),**

-> **lname** **VARCHAR(30)**

-> **)**

-> **PARTITION** **BY** **RANGE** **(id)**

-> **SUBPARTITION** **BY** **KEY** **(lname)**

-> **SUBPARTITIONS** **2** **(**

-> **PARTITION** **p0** **VALUES** **LESS** **THAN** **(50),**

-> **PARTITION** **p1** **VALUES** **LESS** **THAN** **(100),**

-> **PARTITION** **p2** **VALUES** **LESS** **THAN** **(150),**

-> **PARTITION** **p3** **VALUES** **LESS** **THAN** **(MAXVALUE)**

-> **);**

Query OK, 0 rows affected (2.76 sec)

mysql> **INSERT** **INTO** **es** **VALUES**

-> **(1669,** **"Jim",** **"Smith"),**

-> **(337,** **"Mary",** **"Jones"),**

-> **(16,** **"Frank",** **"White"),**

-> **(2005,** **"Linda",** **"Black");**

Query OK, 4 rows affected (0.04 sec)

Records: 4 Duplicates: 0 Warnings: [0](#_bookmark1)

mysql> **CREATE** **TABLE** **es2** **LIKE** **es;**

Query OK, 0 rows affected (1.27 sec)

mysql> **ALTER** **TABLE** **es2** **REMOVE** **PARTITIONING;**

Query OK, 0 rows affected (0.70 sec)

Records: 0 Duplicates: 0 Warnings: [0](#_bookmark2)

Although we did not explicitly name any of the subpartitions when creating table es, we can obtain generated names for these by including the SUBPARTITION\_NAME column of the [PARTITIONS](#_bookmark3) table from INFORMATION\_SCHEMA when selecting from that table, as shown here:

mysql> **SELECT** **PARTITION\_NAME,** **SUBPARTITION\_NAME,** **TABLE\_ROWS**

|  |  |
| --- | --- |
| ->  -> | **FROM** **INFORMATION\_SCHEMA.PARTITIONS**  **WHERE** **TABLE\_NAME** **=** **'es';** |

+----------------+-------------------+------------+

| PARTITION\_NAME | SUBPARTITION\_NAME | TABLE\_ROWS |

+----------------+-------------------+------------+

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | p0  p0  p1  p1  p2  p2  p3  p3 | |  |  |  |  |  |  |  | | p0sp0  p0sp1  p1sp0  p1sp1  p2sp0  p2sp1  p3sp0  p3sp1 | |  |  |  |  |  |  |  | | 1  0  0  0  0  0  3  0 | |  |  |  |  |  |  |  | |

+----------------+-------------------+------------+

8 rows in set (0.00 sec)

The following ALTER TABLE statement exchanges subpartition p3sp0 in table es with the nonpartitioned table es2:

mysql> **ALTER** **TABLE** **es** **EXCHANGE** **PARTITION** **p3sp0** **WITH** **TABLE** **es2;**

Query OK, 0 rows affected (0.29 sec)

You can verify that the rows were exchanged by issuing the following queries:

mysql> **SELECT** **PARTITION\_NAME,** **SUBPARTITION\_NAME,** **TABLE\_ROWS**

|  |  |
| --- | --- |
| ->  -> | **FROM** **INFORMATION\_SCHEMA.PARTITIONS**  **WHERE** **TABLE\_NAME** **=** **'es';** |

+----------------+-------------------+------------+

| PARTITION\_NAME | SUBPARTITION\_NAME | TABLE\_ROWS |

+----------------+-------------------+------------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| | p0  | p0  | p1  | p1  | p2  | p2 | | p0sp0  | p0sp1  | p1sp0  | p1sp1  | p2sp0  | p2sp1 | |  |  |  |  |  | | 1  0  0  0  0  0 | |  |  |  |  |  | |

| p3sp0

| p3

| p3

|

|

|

|

0

0

| p3sp1

+----------------+-------------------+------------+

8 rows in set (0.00 sec)

mysql> **SELECT** **\*** **FROM** **es2;**

+------+-------+-------+

| id | fname | lname |

+------+-------+-------+

| 1669 | Jim | Smith |

| 337 | Mary | Jones |

| 2005 | Linda | Black |

+------+-------+-------+

3 rows in set (0.00 sec)

If a table is subpartitioned, you can exchange only a subpartition of the table—not an entire partition— with an unpartitioned table, as shown here:

mysql> **ALTER** **TABLE** **es** **EXCHANGE** **PARTITION** **p3** **WITH** **TABLE** **es2;**

ERROR 1704 (HY000): Subpartitioned table, use subpartition instead of partition

Table structures are compared in a strict fashion; the number, order, names, and types of columns and indexes of the partitioned table and the nonpartitioned table must match exactly. In addition, both tables must use the same storage engine:

mysql> **CREATE** **TABLE** **es3** **LIKE** **e;**

Query OK, 0 rows affected (1.31 sec)

mysql> **ALTER** **TABLE** **es3** **REMOVE** **PARTITIONING;**

Query OK, 0 rows affected (0.53 sec)

Records: 0 Duplicates: 0 Warnings: [0](#_bookmark4)

mysql> **SHOW** **CREATE** **TABLE** **es3\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Table: es3

Create Table: CREATE TABLE `es3` (

`id` int(11) NOT NULL,

`fname` varchar(30) DEFAULT NULL,

`lname` varchar(30) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4

1 row in set (0.00 sec)

mysql> **ALTER** **TABLE** **es3** **ENGINE** **=** **MyISAM;**

Query OK, 0 rows affected (0.15 sec)

Records: 0 Duplicates: 0 Warnings: [0](#_bookmark5)

mysql> **ALTER** **TABLE** **es** **EXCHANGE** **PARTITION** **p3sp0** **WITH** **TABLE** **es3;**

ERROR 1497 (HY000): The mix of handlers in the partitions is not allowed in this version of MySQL

**24.3.4** **Maintenance** **of** **Partitions**

A number of table and partition maintenance tasks can be carried out on partitioned tables using SQL statements intended for such purposes.

Table maintenance of partitioned tables can be accomplished using the statements CHECK TABLE, OPTIMIZE TABLE, ANALYZE TABLE, and REPAIR TABLE, which are supported for partitioned tables.

You can use a number of extensions to ALTER TABLE for performing operations of this type on one or more partitions directly, as described in the following list:

• **Rebuilding** **partitions.** Rebuilds the partition; this has the same effect as dropping all records stored in the partition, then reinserting them. This can be useful for purposes of defragmentation.

Example:

ALTER TABLE t1 REBUILD PARTITION p0, p1;

• **Optimizing** **partitions.** If you have deleted a large number of rows from a partition or if you have made many changes to a partitioned table with variable-length rows (that is, having VARCHAR, BLOB,

or TEXT columns), you can use ALTER TABLE ... OPTIMIZE PARTITION to reclaim any unused space and to defragment the partition data file.

Example:

ALTER TABLE t1 OPTIMIZE PARTITION p0, p1;

Using OPTIMIZE PARTITION on a given partition is equivalent to running CHECK PARTITION, ANALYZE PARTITION, and REPAIR PARTITION on that partition.

Some MySQL storage engines, including InnoDB, do not support per-partition optimization; in these cases, ALTER TABLE ... OPTIMIZE PARTITION analyzes and rebuilds the entire table, and causes an appropriate warning to be issued. (Bug #11751825, Bug #42822) Use ALTER TABLE ... REBUILD PARTITION and ALTER TABLE ... ANALYZE PARTITION instead, to avoid this issue.

• **Analyzing** **partitions.** This reads and stores the key distributions for partitions.

Example:

ALTER TABLE t1 ANALYZE PARTITION p3;

• **Repairing** **partitions.** This repairs corrupted partitions.

Example:

ALTER TABLE t1 REPAIR PARTITION p0,p1;

Normally, REPAIR PARTITION fails when the partition contains duplicate key errors. You can use ALTER IGNORE TABLE with this option, in which case all rows that cannot be moved due to the presence of duplicate keys are removed from the partition (Bug #16900947).

• **Checking** **partitions.** You can check partitions for errors in much the same way that you can use CHECK TABLE with nonpartitioned tables.

Example:

ALTER TABLE trb3 CHECK PARTITION p1;

This statement tells you whether the data or indexes in partition p1 of table t1 are corrupted. If this is the case, use ALTER TABLE ... REPAIR PARTITION to repair the partition.

Normally, CHECK PARTITION fails when the partition contains duplicate key errors. You can use ALTER IGNORE TABLE with this option, in which case the statement returns the contents of each row in the partition where a duplicate key violation is found. Only the values for the columns in the partitioning expression for the table are reported. (Bug #16900947)

Each of the statements in the list just shown also supports the keyword ALL in place of the list of partition names. Using ALL causes the statement to act on all partitions in the table.

You can also truncate partitions using ALTER TABLE ... TRUNCATE PARTITION. This statement can be used to delete all rows from one or more partitions in much the same way that TRUNCATE TABLE deletes all rows from a table.

ALTER TABLE ... TRUNCATE PARTITION ALL truncates all partitions in the table.

**24.3.5** **Obtaining** **Information** **About** **Partitions**

This section discusses obtaining information about existing partitions, which can be done in a number of ways. Methods of obtaining such information include the following:

• Using the SHOW CREATE TABLE statement to view the partitioning clauses used in creating a partitioned table.

• Using the SHOW TABLE STATUS statement to determine whether a table is partitioned.

• Querying the Information Schema [PARTITIONS](#_bookmark3) table.

• Using the statement EXPLAIN SELECT to see which partitions are used by a given SELECT.

From MySQL 8.0.16, when insertions, deletions, or updates are made to partitioned tables, the binary log records information about the partition and (if any) the subpartition in which the row event took place. A new row event is created for a modification that takes place in a different partition or subpartition, even if the table involved is the same. So if a transaction involves three partitions or subpartitions, three row events are generated. For an update event, the partition information is recorded for both the “before” image and the “after” image. The partition information is displayed if you specify the -v or --verbose option when viewing the binary log using mysqlbinlog. Partition information is only recorded when row-based logging is in use (binlog\_format=ROW).

As discussed elsewhere in this chapter, SHOW CREATE TABLE includes in its output the PARTITION BY clause used to create a partitioned table. For example:

mysql> **SHOW** **CREATE** **TABLE** **trb3\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Table: trb3

Create Table: CREATE TABLE `trb3` (

`id` int(11) DEFAULT NULL,

`name` varchar(50) DEFAULT NULL,

`purchased` date DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4

/\*!50100 PARTITION BY RANGE (YEAR(purchased))

(PARTITION p0 VALUES LESS THAN (1990) ENGINE = InnoDB,

PARTITION p1 VALUES LESS THAN (1995) ENGINE = InnoDB,

PARTITION p2 VALUES LESS THAN (2000) ENGINE = InnoDB,

PARTITION p3 VALUES LESS THAN (2005) ENGINE = InnoDB) \*/

0 row in set (0.00 sec)

The output from SHOW TABLE STATUS for partitioned tables is the same as that for nonpartitioned tables, except that the Create\_options column contains the string partitioned. The Engine column contains the name of the storage engine used by all partitions of the table. (See Section 13.7.7.38, “SHOW TABLE STATUS Statement” , for more information about this statement.)

You can also obtain information about partitions from INFORMATION\_SCHEMA, which contains a [PARTITIONS](#_bookmark3) table. See [Section 26.3.21, “The INFORMATION\_SCHEMA PARTITIONS Table”](#_bookmark3) .

It is possible to determine which partitions of a partitioned table are involved in a given SELECT query using EXPLAIN. The partitions column in the EXPLAIN output lists the partitions from which records would be matched by the query.

Suppose that a table trb1 is created and populated as follows:

CREATE TABLE trb1 (id INT, name VARCHAR(50), purchased DATE)

PARTITION BY RANGE(id)

(

PARTITION p0 VALUES LESS THAN (3),

PARTITION p1 VALUES LESS THAN (7),

PARTITION p2 VALUES LESS THAN (9),

PARTITION p3 VALUES LESS THAN (11)

);

INSERT INTO trb1 VALUES

(1, 'desk organiser', '2003-10-15'),

(2, 'CD player', '1993-11-05'),

(3, 'TV set', '1996-03-10'),

(4, 'bookcase', '1982-01-10'),

(5, 'exercise bike', '2004-05-09'),

(6, 'sofa', '1987-06-05'),

(7, 'popcorn maker', '2001-11-22'),

(8, 'aquarium', '1992-08-04'),

(9, 'study desk', '1984-09-16'),

(10, 'lava lamp', '1998-12-25');

You can see which partitions are used in a query such as SELECT \* FROM trb1;, as shown here:

mysql> **EXPLAIN** **SELECT** **\*** **FROM** **trb1\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: trb1

partitions: p0,p1,p2,p3

type: ALL

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: 10

Extra: Using filesort

In this case, all four partitions are searched. However, when a limiting condition making use of the partitioning key is added to the query, you can see that only those partitions containing matching values are scanned, as shown here:

mysql> **EXPLAIN** **SELECT** **\*** **FROM** **trb1** **WHERE** **id** **<** **5\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: trb1

partitions: p0,p1

type: ALL

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: 10

Extra: Using where

EXPLAIN also provides information about keys used and possible keys:

mysql> **ALTER** **TABLE** **trb1** **ADD** **PRIMARY** **KEY** **(id);**

Query OK, 10 rows affected (0.03 sec)

Records: 10 Duplicates: 0 Warnings: [0](#_bookmark6)

mysql> **EXPLAIN** **SELECT** **\*** **FROM** **trb1** **WHERE** **id** **<** **5\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: trb1

partitions: p0,p1

type: range

possible\_keys: PRIMARY

key: PRIMARY

key\_len: 4

ref: NULL

rows: 7

Extra: Using where

If EXPLAIN is used to examine a query against a nonpartitioned table, no error is produced, but the value of the partitions column is always NULL.

The rows column of EXPLAIN output displays the total number of rows in the table.

See also Section 13.8.2, “EXPLAIN Statement” .

**24.4** **Partition** **Pruning**

The optimization known as *partition* *pruning* is based on a relatively simple concept which can be described as “Do not scan partitions where there can be no matching values” . Suppose a partitioned table t1 is created by this statement:

CREATE TABLE t1 (

fname VARCHAR(50) NOT NULL,

lname VARCHAR(50) NOT NULL,

region\_code TINYINT UNSIGNED NOT NULL,

dob DATE NOT NULL

)

PARTITION BY RANGE( region\_code ) (

PARTITION p0 VALUES LESS THAN (64),

PARTITION p1 VALUES LESS THAN (128),

PARTITION p2 VALUES LESS THAN (192),

PARTITION p3 VALUES LESS THAN MAXVALUE

);

Suppose that you wish to obtain results from a SELECT statement such as this one:

SELECT fname, lname, region\_code, dob

FROM t1

WHERE region\_code > 125 AND region\_code < 130;

It is easy to see that none of the rows which ought to be returned are in either of the partitions p0 or

p3; that is, we need search only in partitions p1 and p2 to find matching rows. By limiting the search, it is possible to expend much less time and effort in finding matching rows than by scanning all partitions

in the table. This “cutting away” of unneeded partitions is known as *pruning*. When the optimizer can make use of partition pruning in performing this query, execution of the query can be an order of magnitude faster than the same query against a nonpartitioned table containing the same column definitions and data.

The optimizer can perform pruning whenever a WHERE condition can be reduced to either one of the following two cases:

• *partition\_column* = *constant*

• *partition\_column* IN (*constant1*, *constant2*, ..., *constantN*)

In the first case, the optimizer simply evaluates the partitioning expression for the value given, determines which partition contains that value, and scans only this partition. In many cases, the equal sign can be replaced with another arithmetic comparison, including <, >, <=, >=, and <>. Some queries using BETWEEN in the WHERE clause can also take advantage of partition pruning. See the examples later in this section.

In the second case, the optimizer evaluates the partitioning expression for each value in the list, creates a list of matching partitions, and then scans only the partitions in this partition list.

SELECT, DELETE, and UPDATE statements support partition pruning. An INSERT statement also accesses only one partition per inserted row; this is true even for a table that is partitioned by HASH or KEY although this is not currently shown in the output of EXPLAIN.

Pruning can also be applied to short ranges, which the optimizer can convert into equivalent lists of values. For instance, in the previous example, the WHERE clause can be converted to WHERE region\_code IN (126, 127, 128, 129). Then the optimizer can determine that the first two values in the list are found in partition p1, the remaining two values in partition p2, and that the other partitions contain no relevant values and so do not need to be searched for matching rows.

The optimizer can also perform pruning for WHERE conditions that involve comparisons of the preceding types on multiple columns for tables that use RANGE COLUMNS or LIST COLUMNS partitioning.

This type of optimization can be applied whenever the partitioning expression consists of an equality or a range which can be reduced to a set of equalities, or when the partitioning expression represents an increasing or decreasing relationship. Pruning can also be applied for tables partitioned on a DATE or DATETIME column when the partitioning expression uses the YEAR() or TO\_DAYS() function. Pruning can also be applied for such tables when the partitioning expression uses the TO\_SECONDS() function.

Suppose that table t2, partitioned on a DATE column, is created using the statement shown here:



CREATE TABLE t2 (

fname VARCHAR(50) NOT NULL,

lname VARCHAR(50) NOT NULL,

region\_code TINYINT UNSIGNED NOT NULL,

dob DATE NOT NULL

)

PARTITION BY RANGE( YEAR(dob) ) (

PARTITION d0 VALUES LESS THAN (1970),

PARTITION d1 VALUES LESS THAN (1975),

PARTITION d2 VALUES LESS THAN (1980),

PARTITION d3 VALUES LESS THAN (1985),

PARTITION d4 VALUES LESS THAN (1990),

PARTITION d5 VALUES LESS THAN (2000),

PARTITION d6 VALUES LESS THAN (2005),

PARTITION d7 VALUES LESS THAN MAXVALUE

);

The following statements using t2 can make of use partition pruning:

SELECT \* FROM t2 WHERE dob = '1982-06-23';

UPDATE t2 SET region\_code = 8 WHERE dob BETWEEN '1991-02-15' AND '1997-04-25';

DELETE FROM t2 WHERE dob >= '1984-06-21' AND dob <= '1999-06-21'

In the case of the last statement, the optimizer can also act as follows:

1. *Find* *the* *partition* *containing* *the* *low* *end* *of* *the* *range*. YEAR('1984-06-21') yields the value 1984, which is found in partition d3.

2. *Find* *the* *partition* *containing* *the* *high* *end* *of* *the* *range*. YEAR('1999-06-21') evaluates to 1999, which is found in partition d5.

3. *Scan* *only* *these* *two* *partitions* *and* *any* *partitions* *that* *may* *lie* *between* *them*.

In this case, this means that only partitions d3, d4, and d5 are scanned. The remaining partitions may be safely ignored (and are ignored).

**Important**

Invalid DATE and DATETIME values referenced in the WHERE condition of a statement against a partitioned table are treated as NULL. This means that a query such as SELECT \* FROM *partitioned\_table* WHERE *date\_column* < '2008-12-00' does not return any values (see Bug #40972).

So far, we have looked only at examples using RANGE partitioning, but pruning can be applied with other partitioning types as well.

Consider a table that is partitioned by LIST, where the partitioning expression is increasing or decreasing, such as the table t3 shown here. (In this example, we assume for the sake of brevity that the region\_code column is limited to values between 1 and 10 inclusive.)

CREATE TABLE t3 (

fname VARCHAR(50) NOT NULL,

lname VARCHAR(50) NOT NULL,

region\_code TINYINT UNSIGNED NOT NULL,

dob DATE NOT NULL

)

PARTITION BY LIST(region\_code) (

PARTITION r0 VALUES IN (1, 3),

PARTITION r1 VALUES IN (2, 5, 8),

PARTITION r2 VALUES IN (4, 9),

PARTITION r3 VALUES IN (6, 7, 10)

);



For a statement such as SELECT \* FROM t3 WHERE region\_code BETWEEN 1 AND 3, the optimizer determines in which partitions the values 1, 2, and 3 are found (r0 and r1) and skips the remaining ones (r2 and r3).

For tables that are partitioned by HASH or [LINEAR] KEY, partition pruning is also possible in cases in which the WHERE clause uses a simple = relation against a column used in the partitioning expression. Consider a table created like this:

CREATE TABLE t4 (

fname VARCHAR(50) NOT NULL,

lname VARCHAR(50) NOT NULL,

region\_code TINYINT UNSIGNED NOT NULL,

dob DATE NOT NULL

)

PARTITION BY KEY(region\_code)

PARTITIONS 8;

A statement that compares a column value with a constant can be pruned:

UPDATE t4 WHERE region\_code = 7;

Pruning can also be employed for short ranges, because the optimizer can turn such conditions into IN relations. For example, using the same table t4 as defined previously, queries such as these can be pruned:

SELECT \* FROM t4 WHERE region\_code > 2 AND region\_code < 6;

SELECT \* FROM t4 WHERE region\_code BETWEEN 3 AND 5;

In both these cases, the WHERE clause is transformed by the optimizer into WHERE region\_code IN

(3, 4, 5).

**Important**

This optimization is used only if the range size is smaller than the number of partitions. Consider this statement:

DELETE FROM t4 WHERE region\_code BETWEEN 4 AND 12;

The range in the WHERE clause covers 9 values (4, 5, 6, 7, 8, 9, 10, 11, 12), but t4 has only 8 partitions. This means that the DELETE cannot be pruned.

When a table is partitioned by HASH or [LINEAR] KEY, pruning can be used only on integer columns. For example, this statement cannot use pruning because dob is a DATE column:

SELECT \* FROM t4 WHERE dob >= '2001-04-14' AND dob <= '2005-10-15';

However, if the table stores year values in an INT column, then a query having WHERE year\_col >=

2001 AND year\_col <= 2005 can be pruned.

Tables using a storage engine that provides automatic partitioning, such as the NDB storage engine used by MySQL Cluster can be pruned if they are explicitly partitioned.

**24.5** **Partition** **Selection**

Explicit selection of partitions and subpartitions for rows matching a given WHERE condition is supported. Partition selection is similar to partition pruning, in that only specific partitions are checked for matches, but differs in two key respects:

1. The partitions to be checked are specified by the issuer of the statement, unlike partition pruning, which is automatic.

2. Whereas partition pruning applies only to queries, explicit selection of partitions is supported for both queries and a number of DML statements.

SQL statements supporting explicit partition selection are listed here:

• SELECT

• DELETE

• INSERT

• REPLACE

• UPDATE

• LOAD DATA.

• LOAD XML.

The remainder of this section discusses explicit partition selection as it applies generally to the statements just listed, and provides some examples.

Explicit partition selection is implemented using a PARTITION option. For all supported statements, this option uses the syntax shown here:

PARTITION (*partition\_names*)

*partition\_names*:

*partition\_name*, ...

This option always follows the name of the table to which the partition or partitions belong. *partition\_names* is a comma-separated list of partitions or subpartitions to be used. Each

name in this list must be the name of an existing partition or subpartition of the specified table; if any of the partitions or subpartitions are not found, the statement fails with an error (partition '*partition\_name* ' doesn't exist). Partitions and subpartitions named in *partition\_names* may be listed in any order, and may overlap.

When the PARTITION option is used, only the partitions and subpartitions listed are checked for matching rows. This option can be used in a SELECT statement to determine which rows belong to a given partition. Consider a partitioned table named employees, created and populated using the statements shown here:

SET @@SQL\_MODE = '';

CREATE TABLE employees (

id INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

fname VARCHAR(25) NOT NULL,

lname VARCHAR(25) NOT NULL,

store\_id INT NOT NULL,

department\_id INT NOT NULL

)

PARTITION BY RANGE(id) (

PARTITION p0 VALUES LESS THAN (5),

PARTITION p1 VALUES LESS THAN (10),

PARTITION p2 VALUES LESS THAN (15),

PARTITION p3 VALUES LESS THAN MAXVALUE

);

INSERT INTO employees VALUES

('', 'Bob', 'Taylor', 3, 2), ('', 'Frank', 'Williams', 1, 2),

('', 'Ellen', 'Johnson', 3, 4), ('', 'Jim', 'Smith', 2, 4),

('', 'Mary', 'Jones', 1, 1), ('', 'Linda', 'Black', 2, 3),

('', 'Ed', 'Jones', 2, 1), ('', 'June', 'Wilson', 3, 1),

('', 'Andy', 'Smith', 1, 3), ('', 'Lou', 'Waters', 2, 4),

('', 'Jill', 'Stone', 1, 4), ('', 'Roger', 'White', 3, 2),

('', 'Howard', 'Andrews', 1, 2), ('', 'Fred', 'Goldberg', 3, 3),

('', 'Barbara', 'Brown', 2, 3), ('', 'Alice', 'Rogers', 2, 2),

('', 'Mark', 'Morgan', 3, 3), ('', 'Karen', 'Cole', 3, 2);

You can see which rows are stored in partition p1 like this:

mysql> **SELECT** **\*** **FROM** **employees** **PARTITION** **(p1);**

+----+-------+--------+----------+---------------+

| id | fname | lname | store\_id | department\_id |

+----+-------+--------+----------+---------------+

| 5 | Mary | Jones | 1 | 1 |

| 6 | Linda | Black | 2 | 3 |

| 7 | Ed | Jones | 2 | 1 |

| 8 | June | Wilson | 3 | 1 |

| 9 | Andy | Smith | 1 | 3 |

+----+-------+--------+----------+---------------+

5 rows in set (0.00 sec)

The result is the same as obtained by the query SELECT \* FROM employees WHERE id BETWEEN

5 AND 9.

To obtain rows from multiple partitions, supply their names as a comma-delimited list. For example, SELECT \* FROM employees PARTITION (p1, p2) returns all rows from partitions p1 and p2 while excluding rows from the remaining partitions.

Any valid query against a partitioned table can be rewritten with a PARTITION option to restrict the result to one or more desired partitions. You can use WHERE conditions, ORDER BY and LIMIT options, and so on. You can also use aggregate functions with HAVING and GROUP BY options. Each of the following queries produces a valid result when run on the employees table as previously defined:

mysql> **SELECT** **\*** **FROM** **employees** **PARTITION** **(p0,** **p2)**

-> **WHERE** **lname** **LIKE** **'S%';**

+----+-------+-------+----------+---------------+

| id | fname | lname | store\_id | department\_id |

+----+-------+-------+----------+---------------+

| 4 | Jim | Smith | 2 | 4 |

| 11 | Jill | Stone | 1 | 4 |

+----+-------+-------+----------+---------------+

2 rows in set (0.00 sec)

mysql> **SELECT** **id,** **CONCAT(fname,** **'** **',** **lname)** **AS** **name**

-> **FROM** **employees** **PARTITION** **(p0)** **ORDER** **BY** **lname;**

+----+----------------+

| id | name |

+----+----------------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  | | 3  4  1  2 | |  |  |  | | Ellen Johnson  Jim Smith  Bob Taylor  Frank Williams | |  |  |  | |

+----+----------------+

4 rows in set (0.06 sec)

mysql> **SELECT** **store\_id,** **COUNT(department\_id)** **AS** **c**

->

->

+---+----------+

| c | store\_id |

+---+----------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | 5  5 | |  | | 2  3 | |  | |

+---+----------+

2 rows in set (0.00 sec)

**FROM** **employees** **PARTITION** **(p1,p2,p3)**

**GROUP** **BY** **store\_id** **HAVING** **c** **>** **4;**

Statements using partition selection can be employed with tables using any of the supported partitioning types. When a table is created using [LINEAR] HASH or [LINEAR] KEY partitioning and the names of the partitions are not specified, MySQL automatically names the partitions p0, p1, p2, ..., p*N-1*, where *N* is the number of partitions. For subpartitions not explicitly named, MySQL assigns automatically to the subpartitions in each partition p*X*the names p*X*sp0, p*X*sp1, p*X*sp2, ..., p*X*sp*M-1*, where *M* is the number of subpartitions. When executing against this table a SELECT (or other SQL statement for which explicit partition selection is allowed), you can use these generated names in a PARTITION option, as shown here:

mysql> **CREATE** **TABLE** **employees\_sub** **(**

-> **id** **INT** **NOT** **NULL** **AUTO\_INCREMENT,**

-> **fname** **VARCHAR(25)** **NOT** **NULL,**

-> **lname** **VARCHAR(25)** **NOT** **NULL,**

-> **store\_id** **INT** **NOT** **NULL,**

-> **department\_id** **INT** **NOT** **NULL,**

-> **PRIMARY** **KEY** **pk** **(id,** **lname)**

-> **)**

-> **PARTITION** **BY** **RANGE(id)**

-> **SUBPARTITION** **BY** **KEY** **(lname)**

-> **SUBPARTITIONS** **2** **(**

-> **PARTITION** **p0** **VALUES** **LESS** **THAN** **(5),**

-> **PARTITION** **p1** **VALUES** **LESS** **THAN** **(10),**

-> **PARTITION** **p2** **VALUES** **LESS** **THAN** **(15),**

-> **PARTITION** **p3** **VALUES** **LESS** **THAN** **MAXVALUE**

-> **);**

Query OK, 0 rows affected (1.14 sec)

mysql> **INSERT** **INTO** **employees\_sub** # reuse data in employees table

-> **SELECT** **\*** **FROM** **employees;**

Query OK, 18 rows affected (0.09 sec)

Records: 18 Duplicates: 0 Warnings: [0](#_bookmark9)

mysql> **SELECT** **id,** **CONCAT(fname,** **'** **',** **lname)** **AS** **name**

-> **FROM** **employees\_sub** **PARTITION** **(p2sp1);**

+----+---------------+

| id | name |

+----+---------------+

| 10 | Lou Waters |

| 14 | Fred Goldberg |

+----+---------------+

2 rows in set (0.00 sec)

You may also use a PARTITION option in the SELECT portion of an INSERT ... SELECT statement, as shown here:

mysql> **CREATE** **TABLE** **employees\_copy** **LIKE** **employees;**

Query OK, 0 rows affected (0.28 sec)

mysql> **INSERT** **INTO** **employees\_copy**

-> **SELECT** **\*** **FROM** **employees** **PARTITION** **(p2);**

Query OK, 5 rows affected (0.04 sec)

Records: 5 Duplicates: 0 Warnings: [0](#_bookmark10)

mysql> **SELECT** **\*** **FROM** **employees\_copy;**

+ +--------+----------+----------+---------------+

| id | fname | lname | store\_id | department\_id |

+----+--------+----------+----------+---------------+

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | 10 | | | Lou | | | Waters | | | 2 | | | 4 | | |
| | | 11 | | | Jill | | | Stone | | | 1 | | | 4 | | |
| | | 12 | | | Roger | | | White | | | 3 | | | 2 | | |
| | | 13 | | | Howard | | | Andrews | | | 1 | | | 2 | | |
| | | 14 | | | Fred | | | Goldberg | | | 3 | | | 3 | | |

+----+--------+----------+----------+---------------+

5 rows in set (0.00 sec)

Partition selection can also be used with joins. Suppose we create and populate two tables using the statements shown here:

CREATE TABLE stores (

id INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

city VARCHAR(30) NOT NULL

)

PARTITION BY HASH(id)

PARTITIONS 2;

INSERT INTO stores VALUES

('', 'Nambucca'), ('', 'Uranga'),

('', 'Bellingen'), ('', 'Grafton');

CREATE TABLE departments (

id INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(30) NOT NULL

)

PARTITION BY KEY(id)

PARTITIONS 2;

INSERT INTO departments VALUES

('', 'Sales'), ('', 'Customer Service'),

('', 'Delivery'), ('', 'Accounting');

You can explicitly select partitions (or subpartitions, or both) from any or all of the tables in a join. (The PARTITION option used to select partitions from a given table immediately follows the name of the table, before all other options, including any table alias.) For example, the following query gets the name, employee ID, department, and city of all employees who work in the Sales or Delivery department (partition p1 of the departments table) at the stores in either of the cities of Nambucca and Bellingen (partition p0 of the stores table):

mysql> **SELECT**

->

->

-> **FROM** **employees** **AS** **e**

|  |  |
| --- | --- |
| ->  -> | **JOIN** **stores** **PARTITION** **(p1)** **AS** **s** **ON** **e.store\_id=s.id**  **JOIN** **departments** **PARTITION** **(p0)** **AS** **d** **ON** **e** **.department\_id=d.id** |

-> **ORDER** **BY** **e** **.lname;**

+-------------+---------------+-----------+------------+

| Employee ID | Name | City | department |

+-------------+---------------+-----------+------------+

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | 14 | | | Fred | Goldberg | | | Bellingen | | | Delivery | | |
| | | 5 | | | Mary | Jones | | | Nambucca | | | Sales | | |
| | | 17 | | | Mark | Morgan | | | Bellingen | | | Delivery | | |
| | | 9 | | | Andy | Smith | | | Nambucca | | | Delivery | | |
| | | 8 | | | June | Wilson | | | Bellingen | | | Sales | | |

+-------------+---------------+-----------+------------+

5 rows in set (0.00 sec)

**e.id** **AS** **'Employee** **ID',** **CONCAT(e.fname,** **'** **',** **e.lname)** **AS** **Name,**

**s.city** **AS** **City,** **d.name** **AS** **department**

For general information about joins in MySQL, see Section 13.2.13.2, “JOIN Clause” .

When the PARTITION option is used with DELETE statements, only those partitions (and subpartitions, if any) listed with the option are checked for rows to be deleted. Any other partitions are ignored, as shown here:

mysql> **SELECT** **\*** **FROM** **employees** **WHERE** **fname** **LIKE** **'j%';**

+----+-------+--------+----------+---------------+

| id | fname | lname | store\_id | department\_id |

+----+-------+--------+----------+---------------+

| 4 | Jim | Smith | 2 | 4 |

| 8 | June | Wilson | 3 | 1 |

| 11 | Jill | Stone | 1 | 4 |

+----+-------+--------+----------+---------------+

3 rows in set (0.00 sec)

mysql> **DELETE** **FROM** **employees** **PARTITION** **(p0,** **p1)**

-> **WHERE** **fname** **LIKE** **'j%';**

Query OK, 2 rows affected (0.09 sec)

mysql> **SELECT** **\*** **FROM** **employees** **WHERE** **fname** **LIKE** **'j%';**

+----+-------+-------+----------+---------------+

| id | fname | lname | store\_id | department\_id |

+----+-------+-------+----------+---------------+

| 11 | Jill | Stone | 1 | 4 |

+----+-------+-------+----------+---------------+

1 row in set (0.00 sec)

Only the two rows in partitions p0 and p1 matching the WHERE condition were deleted. As you can see from the result when the SELECT is run a second time, there remains a row in the table matching the WHERE condition, but residing in a different partition (p2).

UPDATE statements using explicit partition selection behave in the same way; only rows in the partitions referenced by the PARTITION option are considered when determining the rows to be updated, as can be seen by executing the following statements:

mysql> **UPDATE** **employees** **PARTITION** **(p0)**

-> **SET** **store\_id** **=** **2** **WHERE** **fname** **=** **'Jill';**

Query OK, 0 rows affected (0 .00 sec)

Rows matched: 0 Changed: 0 Warnings: 0

mysql> **SELECT** **\*** **FROM** **employees** **WHERE** **fname** **=** **'Jill';**

+ +-------+-------+----------+---------------+

| id | fname | lname | store\_id | department\_id |

+----+-------+-------+----------+---------------+

| 11 | Jill | Stone | 1 | 4 |

+----+-------+-------+----------+---------------+

1 row in set (0.00 sec)

mysql> **UPDATE** **employees** **PARTITION** **(p2)**

-> **SET** **store\_id** **=** **2** **WHERE** **fname** **=** **'Jill';**

Query OK, 1 row affected (0 .09 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> **SELECT** **\*** **FROM** **employees** **WHERE** **fname** **=** **'Jill';**

+ +-------+-------+----------+---------------+

| id | fname | lname | store\_id | department\_id |

+----+-------+-------+----------+---------------+

| 11 | Jill | Stone | 2 | 4 |

+----+-------+-------+----------+---------------+

1 row in set (0.00 sec)

In the same way, when PARTITION is used with DELETE, only rows in the partition or partitions named in the partition list are checked for deletion.

For statements that insert rows, the behavior differs in that failure to find a suitable partition causes the statement to fail. This is true for both INSERT and REPLACE statements, as shown here:

mysql> **INSERT** **INTO** **employees** **PARTITION** **(p2)** **VALUES** **(20,** **'Jan',** **'Jones',** **1,** **3);**

ERROR 1729 (HY000): Found a row not matching the given partition set

mysql> **INSERT** **INTO** **employees** **PARTITION** **(p3)** **VALUES** **(20,** **'Jan',** **'Jones',** **1,** **3);**

Query OK, 1 row affected (0.07 sec)

mysql> REPLACE INTO employees PARTITION (p0) VALUES (20, 'Jan', 'Jones', 3, 2);

ERROR 1729 (HY000): Found a row not matching the given partition set

mysql> REPLACE INTO employees PARTITION (p3) VALUES (20, 'Jan', 'Jones', 3, 2);

Query OK, 2 rows affected (0.09 sec)

For statements that write multiple rows to a partitioned table that using the InnoDB storage engine: If any row in the list following VALUES cannot be written to one of the partitions specified in the *partition\_names* list, the entire statement fails and no rows are written. This is shown for INSERT statements in the following example, reusing the employees table created previously:

mysql> **ALTER** **TABLE** **employees**

-> **REORGANIZE** **PARTITION** **p3** **INTO** **(**

-> **PARTITION** **p3** **VALUES** **LESS** **THAN** **(20),**

-> **PARTITION** **p4** **VALUES** **LESS** **THAN** **(25),**

-> **PARTITION** **p5** **VALUES** **LESS** **THAN** **MAXVALUE**

-> **);**

Query OK, 6 rows affected (2.09 sec)

Records: 6 Duplicates: 0 Warnings: [0](#_bookmark11)

mysql> **SHOW** **CREATE** **TABLE** **employees\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Table: employees

Create Table: CREATE TABLE `employees` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`fname` varchar(25) NOT NULL,

`lname` varchar(25) NOT NULL,

`store\_id` int(11) NOT NULL,

`department\_id` int(11) NOT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=27 DEFAULT CHARSET=utf8mb4

/\*!50100 PARTITION BY RANGE (id)

(PARTITION p0 VALUES LESS THAN (5) ENGINE = InnoDB,

PARTITION p1 VALUES LESS THAN (10) ENGINE = InnoDB,

PARTITION p2 VALUES LESS THAN (15) ENGINE = InnoDB,

PARTITION p3 VALUES LESS THAN (20) ENGINE = InnoDB,

PARTITION p4 VALUES LESS THAN (25) ENGINE = InnoDB,

PARTITION p5 VALUES LESS THAN MAXVALUE ENGINE = InnoDB) \*/

1 row in set (0.00 sec)

mysql> **INSERT** **INTO** **employees** **PARTITION** **(p3,** **p4)** **VALUES**

-> **(24,** **'Tim',** **'Greene',** **3,** **1),** **(26,** **'Linda',** **'Mills',** **2,** **1);**

ERROR 1729 (HY000): Found a row not matching the given partition set

mysql> **INSERT** **INTO** **employees** **PARTITION** **(p3,** **p4,** **p5)** **VALUES**

-> **(24,** **'Tim',** **'Greene',** **3,** **1),** **(26,** **'Linda',** **'Mills',** **2,** **1);**

Query OK, 2 rows affected (0.06 sec)

Records: 2 Duplicates: 0 Warnings: [0](#_bookmark12)

The preceding is true for both INSERT statements and REPLACE statements that write multiple rows.

Partition selection is disabled for tables employing a storage engine that supplies automatic partitioning, such as NDB.

**24.6** **Restrictions** **and** **Limitations** **on** **Partitioning**

This section discusses current restrictions and limitations on MySQL partitioning support. **Prohibited** **constructs.** The following constructs are not permitted in partitioning expressions:

• Stored procedures, stored functions, loadable functions, or plugins.

• Declared variables or user variables.

For a list of SQL functions which are permitted in partitioning expressions, see [Section 24.6.3,](#_bookmark13) [“Partitioning Limitations Relating to Functions”](#_bookmark13) .

**Arithmetic** **and** **logical** **operators.** Use of the arithmetic operators +, -, and \* is permitted in

partitioning expressions. However, the result must be an integer value or NULL (except in the case of [LINEAR] KEY partitioning, as discussed elsewhere in this chapter; see Section 24.2, “Partitioning Types” , for more information).

The DIV operator is also supported; the / operator is not permitted.

The bit operators |, &, ^, <<, >>, and ~ are not permitted in partitioning expressions.

**Server** **SQL** **mode.** Tables employing user-defined partitioning do not preserve the SQL mode in effect at the time that they were created. As discussed elsewhere in this Manual (see Section 5.1.11, “Server SQL Modes”), the results of many MySQL functions and operators may change according to the server SQL mode. Therefore, a change in the SQL mode at any time after the creation of partitioned tables may lead to major changes in the behavior of such tables, and could easily lead to corruption or loss of data. For these reasons, *it* *is* *strongly* *recommended* *that* *you* *never* *change* *the* *server* *SQL* *mode* *after* *creating* *partitioned* *tables*.

For one such change in the server SQL mode making a partitioned tables unusable, consider the following CREATE TABLE statement, which can be executed successfully only if the NO\_UNSIGNED\_SUBTRACTION mode is in effect:

mysql> **SELECT** **@@sql\_mode;**

+------------+

| @@sql\_mode |

+------------+

| |

+------------+

1 row in set (0.00 sec)

mysql> **CREATE** **TABLE** **tu** **(c1** **BIGINT** **UNSIGNED)**

-> **PARTITION** **BY** **RANGE(c1** **-** **10)** **(**

-> **PARTITION** **p0** **VALUES** **LESS** **THAN** **(-5),**

-> **PARTITION** **p1** **VALUES** **LESS** **THAN** **(0),**

-> **PARTITION** **p2** **VALUES** **LESS** **THAN** **(5),**

-> **PARTITION** **p3** **VALUES** **LESS** **THAN** **(10),**

-> **PARTITION** **p4** **VALUES** **LESS** **THAN** **(MAXVALUE)**

-> **);**

ERROR 1563 (HY000): Partition constant is out of partition function domain

mysql> **SET** **sql\_mode='NO\_UNSIGNED\_SUBTRACTION';**

Query OK, 0 rows affected (0.00 sec)

mysql> **SELECT** **@@sql\_mode;**

+-------------------------+

| @@sql\_mode |

+-------------------------+

| NO\_UNSIGNED\_SUBTRACTION |

+-------------------------+

1 row in set (0.00 sec)

mysql> **CREATE** **TABLE** **tu** **(c1** **BIGINT** **UNSIGNED)**

-> **PARTITION** **BY** **RANGE(c1** **-** **10)** **(**

-> **PARTITION** **p0** **VALUES** **LESS** **THAN** **(-5),**

-> **PARTITION** **p1** **VALUES** **LESS** **THAN** **(0),**

-> **PARTITION** **p2** **VALUES** **LESS** **THAN** **(5),**

-> **PARTITION** **p3** **VALUES** **LESS** **THAN** **(10),**

-> **PARTITION** **p4** **VALUES** **LESS** **THAN** **(MAXVALUE)**

-> **);**

Query OK, 0 rows affected (0.05 sec)

If you remove the NO\_UNSIGNED\_SUBTRACTION server SQL mode after creating tu, you may no longer be able to access this table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| mysql> **SET** **sql\_mode='';**  Query OK, 0 rows affected (0.00 sec)  mysql> **SELECT** **\*** **FROM** **tu;**  ERROR 1563 (HY000): Partition constant  mysql> **INSERT** **INTO** **tu** **VALUES** **(20);**  ERROR 1563 (HY000): Partition constant | is  is | out of  out of | partition  partition | function domain  function domain |

See also Section 5.1.11, “Server SQL Modes” .

Server SQL modes also impact replication of partitioned tables. Disparate SQL modes on source and replica can lead to partitioning expressions being evaluated differently; this can cause the distribution of data among partitions to be different in the source's and replica's copies of a given table, and may even cause inserts into partitioned tables that succeed on the source to fail on the replica. For best results, you should always use the same server SQL mode on the source and on the replica.

**Performance** **considerations.** Some effects of partitioning operations on performance are given in

the following list:

• **File** **system** **operations.** Partitioning and repartitioning operations (such as ALTER TABLE with PARTITION BY ..., REORGANIZE PARTITION, or REMOVE PARTITIONING) depend on

file system operations for their implementation. This means that the speed of these operations is affected by such factors as file system type and characteristics, disk speed, swap space, file handling efficiency of the operating system, and MySQL server options and variables that relate to file handling. In particular, you should make sure that large\_files\_support is enabled and that open\_files\_limit is set properly. Partitioning and repartitioning operations involving InnoDB tables may be made more efficient by enabling innodb\_file\_per\_table.

See also [Maximum number of partitions](#_bookmark15).

• **Table** **locks.** Generally, the process executing a partitioning operation on a table takes a write lock on the table. Reads from such tables are relatively unaffected; pending INSERT and UPDATE operations are performed as soon as the partitioning operation has completed. For InnoDB-specific exceptions to this limitation, see Partitioning Operations.

• **Indexes;** **partition** **pruning.** As with nonpartitioned tables, proper use of indexes can speed up queries on partitioned tables significantly. In addition, designing partitioned tables and queries

on these tables to take advantage of *partition* *pruning* can improve performance dramatically. See [Section 24.4, “Partition Pruning”](#_bookmark7) , for more information.

Index condition pushdown is supported for partitioned tables. See Section 8.2.1.6, “Index Condition Pushdown Optimization” .

• **Performance** **with** **LOAD** **DATA.** In MySQL 8.0, LOAD DATA uses buffering to improve performance. You should be aware that the buffer uses 130 KB memory per partition to achieve this.

**Maximum** **number** **of** **partitions.**

The maximum possible number of partitions for a given table not using the NDB storage engine is 8192. This number includes subpartitions.

The maximum possible number of user-defined partitions for a table using the NDB storage engine is determined according to the version of the NDB Cluster software being used, the number of data nodes, and other factors. See NDB and user-defined partitioning, for more information.

If, when creating tables with a large number of partitions (but less than the maximum), you encounter an error message such as Got error ... from storage engine: Out of resources when opening file, you may be able to address the issue by increasing the value of the open\_files\_limit system variable. However, this is dependent on the operating system, and may not be possible or advisable on all platforms; see Section B.3.2.16, “File Not Found and Similar Errors” , for more information. In some cases, using large numbers (hundreds) of partitions may also not be advisable due to other concerns, so using more partitions does not automatically lead to better results.

See also [File system operations](#_bookmark14).

**Foreign** **keys** **not** **supported** **for** **partitioned** **InnoDB** **tables.**

Partitioned tables using the InnoDB storage engine do not support foreign keys. More specifically, this means that the following two statements are true:

1. No definition of an InnoDB table employing user-defined partitioning may contain foreign key references; no InnoDB table whose definition contains foreign key references may be partitioned.

2. No InnoDB table definition may contain a foreign key reference to a user-partitioned table; no InnoDB table with user-defined partitioning may contain columns referenced by foreign keys.

The scope of the restrictions just listed includes all tables that use the InnoDB storage engine. CREATE TABLE and ALTER TABLE statements that would result in tables violating these restrictions are not allowed.

**ALTER** **TABLE** **...** **ORDER** **BY.** An ALTER TABLE ... ORDER BY *column* statement run against a partitioned table causes ordering of rows only within each partition.

**ADD** **COLUMN** **...** **ALGORITHM=INSTANT.** Once you perform ALTER TABLE ... ADD COLUMN ... ALGORITHM=INSTANT on a partitioned table, it is no longer possible to exchange partitions with this table.

**Effects** **on** **REPLACE** **statements** **by** **modification** **of** **primary** **keys.** It can be desirable in some cases (see [Section 24.6.1, “Partitioning Keys, Primary Keys, and Unique Keys”](#_bookmark16)) to modify a table's primary key. Be aware that, if your application uses REPLACE statements and you do this, the results of these statements can be drastically altered. See Section 13.2.12, “REPLACE Statement” , for more information and an example.

**FULLTEXT** **indexes.**

Partitioned tables do not support FULLTEXT indexes or searches.

**Spatial** **columns.** Columns with spatial data types such as POINT or GEOMETRY cannot be used in

partitioned tables.

**Temporary** **tables.**

Temporary tables cannot be partitioned.

**Log** **tables.** It is not possible to partition the log tables; an ALTER TABLE ... PARTITION BY ... statement on such a table fails with an error.

**Data** **type** **of** **partitioning** **key.**

A partitioning key must be either an integer column or an expression that resolves to an integer. Expressions employing ENUM columns cannot be used. The column or expression value may also be NULL; see Section 24.2.7, “How MySQL Partitioning Handles NULL” .

There are two exceptions to this restriction:

1. When partitioning by [LINEAR] KEY, it is possible to use columns of any valid MySQL data type other than TEXT or BLOB as partitioning keys, because the internal key-hashing functions produce the correct data type from these types. For example, the following two CREATE TABLE statements are valid:

CREATE TABLE tkc (c1 CHAR)

PARTITION BY KEY(c1)

PARTITIONS 4;

CREATE TABLE tke

( c1 ENUM('red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet') )

PARTITION BY LINEAR KEY(c1)

PARTITIONS 6;

2. When partitioning by RANGE COLUMNS or LIST COLUMNS, it is possible to use string, DATE, and

DATETIME columns. For example, each of the following CREATE TABLE statements is valid:

CREATE TABLE rc (c1 INT, c2 DATE)

PARTITION BY RANGE COLUMNS(c2) (

PARTITION p0 VALUES LESS THAN('1990-01-01'),

PARTITION p1 VALUES LESS THAN('1995-01-01'),

PARTITION p2 VALUES LESS THAN('2000-01-01'),

PARTITION p3 VALUES LESS THAN('2005-01-01'),

PARTITION p4 VALUES LESS THAN(MAXVALUE)

);

CREATE TABLE lc (c1 INT, c2 CHAR(1))

PARTITION BY LIST COLUMNS(c2) (

PARTITION p0 VALUES IN('a', 'd', 'g', 'j', 'm', 'p', 's', 'v', 'y'),

PARTITION p1 VALUES IN('b', 'e', 'h', 'k', 'n', 'q', 't', 'w', 'z'),

PARTITION p2 VALUES IN('c', 'f', 'i', 'l', 'o', 'r', 'u', 'x', NULL)

);

Neither of the preceding exceptions applies to BLOB or TEXT column types.

**Subqueries.**

A partitioning key may not be a subquery, even if that subquery resolves to an integer value or NULL.

**Column** **index** **prefixes** **not** **supported** **for** **key** **partitioning.** When creating a table that is partitioned by key, any columns in the partitioning key which use column prefixes are not used in the table's partitioning function. Consider the following CREATE TABLE statement, which has three VARCHAR columns, and whose primary key uses all three columns and specifies prefixes for two of them:

CREATE TABLE t1 (

a VARCHAR(10000),

b VARCHAR(25),

c VARCHAR(10),

PRIMARY KEY (a(10), b, c(2))

) PARTITION BY KEY() PARTITIONS 2;

This statement is accepted, but the resulting table is actually created as if you had issued the following statement, using only the primary key column which does not include a prefix (column b) for the partitioning key:

CREATE TABLE t1 (

a VARCHAR(10000),

b VARCHAR(25),

c VARCHAR(10),

PRIMARY KEY (a(10), b, c(2))

) PARTITION BY KEY(b) PARTITIONS 2;

Prior to MySQL 8.0.21, no warning was issued or any other indication provided that this occurred, except in the event that all columns specified for the partitioning key used prefixes, in which case the statement failed, but with a misleading error message, as shown here:

mysql> **CREATE** **TABLE** **t2** **(**

->

->

->

->

-> **)** **PARTITION** **BY** **KEY()** **PARTITIONS** **2;**

ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the

table's partitioning function

**a** **VARCHAR(10000),**

**b** **VARCHAR(25),**

**c** **VARCHAR(10),**

**PRIMARY** **KEY** **(a(10),** **b(5),** **c(2))**

This also occurred when performing ALTER TABLE or when upgrading such tables.

This permissive behavior is deprecated as of MySQL 8.0.21 (and is subject to removal in a future version of MySQL). Beginning with MySQL 8.0.21, using one or more columns having a prefix in the partitioning key results in a warning for each such column, as shown here:

mysql> **CREATE** **TABLE** **t1** **(**

->

->

->

->

-> **)** **PARTITION** **BY** **KEY()** **PARTITIONS** **2;**

Query OK, 0 rows affected, 2 warnings (1.25 sec)

mysql> **SHOW** **WARNINGS\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Level: Warning

Code: 1681

Message: Column 'test.t1.a' having prefix key part 'a(10)' is ignored by the

partitioning function . Use of prefixed columns in the PARTITION BY KEY() clause

is deprecated and will be removed in a future release .

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Level: Warning

Code: 1681

Message: Column 'test.t1.c' having prefix key part 'c(2)' is ignored by the

partitioning function . Use of prefixed columns in the PARTITION BY KEY() clause

is deprecated and will be removed in a future release .

2 rows in set (0.00 sec)

**a** **VARCHAR(10000),**

**b** **VARCHAR(25),**

**c** **VARCHAR(10),**

**PRIMARY** **KEY** **(a(10),** **b,** **c(2))**

This includes cases in which the columns used in the partitioning function are defined implicitly as those in the table's primary key by employing an empty PARTITION BY KEY() clause.

In MySQL 8.0.21 and later, if all columns specified for the partitioning key employ prefixes, the CREATE TABLE statement used fails with an error message that identifies the issue correctly:

mysql> **CREATE** **TABLE** **t1** **(**

->

->

->

->

-> **)** **PARTITION** **BY** **KEY()** **PARTITIONS** **2;**

ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the table's

partitioning function (prefixed columns are not considered).

**a** **VARCHAR(10000),**

**b** **VARCHAR(25),**

**c** **VARCHAR(10),**

**PRIMARY** **KEY** **(a(10),** **b(5),** **c(2))**

For general information about partitioning tables by key, see Section 24.2.5, “KEY Partitioning” .

**Issues** **with** **subpartitions.**

Subpartitions must use HASH or KEY partitioning. Only RANGE and LIST partitions may be subpartitioned; HASH and KEY partitions cannot be subpartitioned.

SUBPARTITION BY KEY requires that the subpartitioning column or columns be specified explicitly, unlike the case with PARTITION BY KEY, where it can be omitted (in which case the table's primary key column is used by default). Consider the table created by this statement:

CREATE TABLE ts (

id INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(30)

);

You can create a table having the same columns, partitioned by KEY, using a statement such as this one:

CREATE TABLE ts (

id INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(30)

)

PARTITION BY KEY()

PARTITIONS 4;

The previous statement is treated as though it had been written like this, with the table's primary key column used as the partitioning column:

CREATE TABLE ts (

id INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(30)

)

PARTITION BY KEY(id)

PARTITIONS 4;

However, the following statement that attempts to create a subpartitioned table using the default column as the subpartitioning column fails, and the column must be specified for the statement to succeed, as shown here:

mysql> **CREATE** **TABLE** **ts** **(**

-> **id** **INT** **NOT** **NULL** **AUTO\_INCREMENT** **PRIMARY** **KEY,**

-> **name** **VARCHAR(30)**

-> **)**

-> **PARTITION** **BY** **RANGE(id)**

-> **SUBPARTITION** **BY** **KEY()**

-> **SUBPARTITIONS** **4**

-> **(**

->

->

-> **);**

ERROR 1064

corresponds to your MySQL server version for the right syntax to use near ')

mysql> **CREATE** **TABLE** **ts** **(**

-> **id** **INT** **NOT** **NULL** **AUTO\_INCREMENT** **PRIMARY** **KEY,**

-> **name** **VARCHAR(30)**

-> **)**

-> **PARTITION** **BY** **RANGE(id)**

-> **SUBPARTITION** **BY** **KEY(id)**

-> **SUBPARTITIONS** **4**

-> **(**

-> **PARTITION** **p0** **VALUES** **LESS** **THAN** **(100),**

-> **PARTITION** **p1** **VALUES** **LESS** **THAN** **(MAXVALUE)**

-> **);**

Query OK, 0 rows affected (0.07 sec)

(42000): You have an error in your SQL syntax; check the manual that

**PARTITION** **p0** **VALUES** **LESS** **THAN** **(100),**

**PARTITION** **p1** **VALUES** **LESS** **THAN** **(MAXVALUE)**

This is a known issue (see Bug #51470).

**DATA** **DIRECTORY** **and** **INDEX** **DIRECTORY** **options.** Table-level DATA DIRECTORY and INDEX DIRECTORY options are ignored (see Bug #32091). You can employ these options for individual partitions or subpartitions of InnoDB tables. As of MySQL 8.0.21, the directory specified in a DATA DIRECTORY clause must be known to InnoDB. For more information, see Using the DATA

DIRECTORY Clause.

**Repairing** **and** **rebuilding** **partitioned** **tables.** The statements CHECK TABLE, OPTIMIZE TABLE, ANALYZE TABLE, and REPAIR TABLE are supported for partitioned tables.

In addition, you can use ALTER TABLE ... REBUILD PARTITION to rebuild one or more partitions of a partitioned table; ALTER TABLE ... REORGANIZE PARTITION also causes partitions to be rebuilt. See Section 13.1.9, “ALTER TABLE Statement” , for more information about these two statements.

ANALYZE, CHECK, OPTIMIZE, REPAIR, and TRUNCATE operations are supported with subpartitions. See Section 13.1.9.1, “ALTER TABLE Partition Operations” .

**File** **name** **delimiters** **for** **partitions** **and** **subpartitions.** Table partition and subpartition file names include generated delimiters such as #P# and #SP#. The lettercase of such delimiters can vary and should not be depended upon.

**24.6.1** **Partitioning** **Keys,** **Primary** **Keys,** **and** **Unique** **Keys**

This section discusses the relationship of partitioning keys with primary keys and unique keys. The rule governing this relationship can be expressed as follows: All columns used in the partitioning expression for a partitioned table must be part of every unique key that the table may have.

In other words, *every* *unique* *key* *on* *the* *table* *must* *use* *every* *column* *in* *the* *table's* *partitioning* *expression*. (This also includes the table's primary key, since it is by definition a unique key. This particular case is discussed later in this section.) For example, each of the following table creation statements is invalid:

CREATE TABLE t1 (

col1 INT NOT NULL,

col2 DATE NOT NULL,

col3 INT NOT NULL,

col4 INT NOT NULL,

UNIQUE KEY (col1, col2)

)

PARTITION BY HASH(col3)

PARTITIONS 4;

CREATE TABLE t2 (

col1 INT NOT NULL,

col2 DATE NOT NULL,

col3 INT NOT NULL,

col4 INT NOT NULL,

UNIQUE KEY (col1),

UNIQUE KEY (col3)

)

PARTITION BY HASH(col1 + col3)

PARTITIONS 4;

In each case, the proposed table would have at least one unique key that does not include all columns used in the partitioning expression.

Each of the following statements is valid, and represents one way in which the corresponding invalid table creation statement could be made to work:

CREATE TABLE t1 (

col1 INT NOT NULL,

col2 DATE NOT NULL,

col3 INT NOT NULL,

col4 INT NOT NULL,

UNIQUE KEY (col1, col2, col3)

)

PARTITION BY HASH(col3)

PARTITIONS 4;

CREATE TABLE t2 (

col1 INT NOT NULL,

col2 DATE NOT NULL,

col3 INT NOT NULL,

col4 INT NOT NULL,

UNIQUE KEY (col1, col3)

)

PARTITION BY HASH(col1 + col3)

PARTITIONS 4;

This example shows the error produced in such cases:

|  |  |
| --- | --- |
| mysql>  ->  ->  ->  ->  ->  ->  ->  ->  ->  ERROR | **CREATE** **TABLE** **t3** **(**  **col1** **INT** **NOT** **NULL,**  **col2** **DATE** **NOT** **NULL,**  **col3** **INT** **NOT** **NULL,**  **col4** **INT** **NOT** **NULL,**  **UNIQUE** **KEY** **(col1,** **col2),**  **UNIQUE** **KEY** **(col3)**  **)**  **PARTITION** **BY** **HASH(col1** **+** **col3)**  **PARTITIONS** **4;**  1491 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function |

The CREATE TABLE statement fails because both col1 and col3 are included in the proposed partitioning key, but neither of these columns is part of both of unique keys on the table. This shows one possible fix for the invalid table definition:

mysql> **CREATE** **TABLE** **t3** **(**

-> **col1** **INT** **NOT** **NULL,**

-> **col2** **DATE** **NOT** **NULL,**

-> **col3** **INT** **NOT** **NULL,**

-> **col4** **INT** **NOT** **NULL,**

-> **UNIQUE** **KEY** **(col1,** **col2,** **col3),**

-> **UNIQUE** **KEY** **(col3)**

-> **)**

-> **PARTITION** **BY** **HASH(col3)**

-> **PARTITIONS** **4;**

Query OK, 0 rows affected (0.05 sec)

In this case, the proposed partitioning key col3 is part of both unique keys, and the table creation statement succeeds.

The following table cannot be partitioned at all, because there is no way to include in a partitioning key any columns that belong to both unique keys:

CREATE TABLE t4 (

col1 INT NOT NULL,

col2 INT NOT NULL,

col3 INT NOT NULL,

col4 INT NOT NULL,

UNIQUE KEY (col1, col3),

UNIQUE KEY (col2, col4)

);

Since every primary key is by definition a unique key, this restriction also includes the table's primary key, if it has one. For example, the next two statements are invalid:

CREATE TABLE t5 (

col1 INT NOT NULL,

col2 DATE NOT NULL,

col3 INT NOT NULL,

col4 INT NOT NULL,

PRIMARY KEY(col1, col2)

)

PARTITION BY HASH(col3)

PARTITIONS 4;

CREATE TABLE t6 (

col1 INT NOT NULL,

col2 DATE NOT NULL,

col3 INT NOT NULL,

col4 INT NOT NULL,

PRIMARY KEY(col1, col3),

UNIQUE KEY(col2)

)

PARTITION BY HASH( YEAR(col2) )

PARTITIONS 4;

In both cases, the primary key does not include all columns referenced in the partitioning expression. However, both of the next two statements are valid:

CREATE TABLE t7 (

col1 INT NOT NULL,

col2 DATE NOT NULL,

col3 INT NOT NULL,

col4 INT NOT NULL,

PRIMARY KEY(col1, col2)

)

PARTITION BY HASH(col1 + YEAR(col2))

PARTITIONS 4;

CREATE TABLE t8 (

col1 INT NOT NULL,

col2 DATE NOT NULL,

col3 INT NOT NULL,

col4 INT NOT NULL,

PRIMARY KEY(col1, col2, col4),

UNIQUE KEY(col2, col1)

)

PARTITION BY HASH(col1 + YEAR(col2))

PARTITIONS 4;

If a table has no unique keys—this includes having no primary key—then this restriction does not apply, and you may use any column or columns in the partitioning expression as long as the column type is compatible with the partitioning type.

For the same reason, you cannot later add a unique key to a partitioned table unless the key includes all columns used by the table's partitioning expression. Consider the partitioned table created as shown here:

mysql> **CREATE** **TABLE** **t\_no\_pk** **(c1** **INT,** **c2** **INT)**

-> **PARTITION** **BY** **RANGE(c1)** **(**

-> **PARTITION** **p0** **VALUES** **LESS** **THAN** **(10),**

-> **PARTITION** **p1** **VALUES** **LESS** **THAN** **(20),**

-> **PARTITION** **p2** **VALUES** **LESS** **THAN** **(30),**

-> **PARTITION** **p3** **VALUES** **LESS** **THAN** **(40)**

-> **);**

Query OK, 0 rows affected (0.12 sec)

It is possible to add a primary key to t\_no\_pk using either of these ALTER TABLE statements:

# possible PK

mysql> **ALTER** **TABLE** **t\_no\_pk** **ADD** **PRIMARY** **KEY(c1);**

Query OK, 0 rows affected (0.13 sec)

Records: 0 Duplicates: 0 Warnings: [0](#_bookmark17)

# drop this PK

mysql> **ALTER** **TABLE** **t\_no\_pk** **DROP** **PRIMARY** **KEY;**

Query OK, 0 rows affected (0.10 sec)

Records: 0 Duplicates: 0 Warnings: [0](#_bookmark18)

# use another possible PK

mysql> **ALTER** **TABLE** **t\_no\_pk** **ADD** **PRIMARY** **KEY(c1,** **c2);**

Query OK, 0 rows affected (0.12 sec)

Records: 0 Duplicates: 0 Warnings: [0](#_bookmark19)

# drop this PK

mysql> **ALTER** **TABLE** **t\_no\_pk** **DROP** **PRIMARY** **KEY;**

Query OK, 0 rows affected (0.09 sec)

Records: 0 Duplicates: 0 Warnings: [0](#_bookmark20)

However, the next statement fails, because c1 is part of the partitioning key, but is not part of the proposed primary key:

# fails with error 1503

mysql> **ALTER** **TABLE** **t\_no\_pk** **ADD** **PRIMARY** **KEY(c2);**

ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function

Since t\_no\_pk has only c1 in its partitioning expression, attempting to adding a unique key on c2 alone fails. However, you can add a unique key that uses both c1 and c2.

These rules also apply to existing nonpartitioned tables that you wish to partition using ALTER TABLE ... PARTITION BY. Consider a table np\_pk created as shown here:

mysql> **CREATE** **TABLE** **np\_pk** **(**

-> **id** **INT** **NOT** **NULL** **AUTO\_INCREMENT,**

-> **name** **VARCHAR(50),**

-> **added** **DATE,**

-> **PRIMARY** **KEY** **(id)**

-> **);**

Query OK, 0 rows affected (0.08 sec)

The following ALTER TABLE statement fails with an error, because the added column is not part of any unique key in the table:

mysql> **ALTER** **TABLE** **np\_pk**

-> **PARTITION** **BY** **HASH(** **TO\_DAYS(added)** **)**

-> **PARTITIONS** **4;**

ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function

However, this statement using the id column for the partitioning column is valid, as shown here:

mysql> **ALTER** **TABLE** **np\_pk**

-> **PARTITION** **BY** **HASH(id)**

-> **PARTITIONS** **4;**

Query OK, 0 rows affected (0.11 sec)

Records: 0 Duplicates: 0 Warnings: [0](#_bookmark17)

In the case of np\_pk, the only column that may be used as part of a partitioning expression is id; if you wish to partition this table using any other column or columns in the partitioning expression, you must first modify the table, either by adding the desired column or columns to the primary key, or by dropping the primary key altogether.

**24.6.2** **Partitioning** **Limitations** **Relating** **to** **Storage** **Engines**

In MySQL 8.0, partitioning support is not actually provided by the MySQL Server, but rather by a table storage engine's own or native partitioning handler. In MySQL 8.0, only the InnoDB and NDB storage engines provide native partitioning handlers. This means that partitioned tables cannot be created using any other storage engine than these. (You must be using MySQL NDB Cluster with the NDB storage engine to create NDB tables.)

**InnoDB** **storage** **engine.** InnoDB foreign keys and MySQL partitioning are not compatible. Partitioned InnoDB tables cannot have foreign key references, nor can they have columns referenced by foreign keys. InnoDB tables which have or which are referenced by foreign keys cannot be partitioned.

ALTER TABLE ... OPTIMIZE PARTITION does not work correctly with partitioned tables that use InnoDB. Use ALTER TABLE ... REBUILD PARTITION and ALTER TABLE ... ANALYZE PARTITION, instead, for such tables. For more information, see Section 13.1.9.1, “ALTER TABLE Partition Operations” .

**User-defined** **partitioning** **and** **the** **NDB** **storage** **engine** **(NDB** **Cluster).** Partitioning by KEY (including LINEAR KEY) is the only type of partitioning supported for the NDB storage engine. It is not possible under normal circumstances in NDB Cluster to create an NDB Cluster table using any partitioning type other than [LINEAR] KEY, and attempting to do so fails with an error.

*Exception* *(not* *for* *production)*: It is possible to override this restriction by setting the new system variable on NDB Cluster SQL nodes to ON. If you choose to do this, you should be aware that tables

using partitioning types other than [LINEAR] KEY are not supported in production. *In* *such* *cases,* *you* *can* *create* *and* *use* *tables* *with* *partitioning* *types* *other* *than* *KEY* *or* *L工NEAR* *KEY,* *but* *you* *do* *this* *entirely* *at* *your* *own* *risk*.

The maximum number of partitions that can be defined for an NDB table depends on the number of data nodes and node groups in the cluster, the version of the NDB Cluster software in use, and other factors. See NDB and user-defined partitioning, for more information.

The maximum amount of fixed-size data that can be stored per partition in an NDB table is 128 TB. Previously, this was 16 GB.

CREATE TABLE and ALTER TABLE statements that would cause a user-partitioned NDB table not to meet either or both of the following two requirements are not permitted, and fail with an error:

1. The table must have an explicit primary key.

2. All columns listed in the table's partitioning expression must be part of the primary key.

**Exception.** If a user-partitioned NDB table is created using an empty column-list (that is, using PARTITION BY KEY() or PARTITION BY LINEAR KEY()), then no explicit primary key is required.

**Partition** **selection.** Partition selection is not supported for NDB tables. See [Section 24.5, “Partition](#_bookmark8) [Selection”](#_bookmark8) , for more information.

**Upgrading** **partitioned** **tables.** When performing an upgrade, tables which are partitioned by KEY must be dumped and reloaded. Partitioned tables using storage engines other than InnoDB cannot be upgraded from MySQL 5.7 or earlier to MySQL 8.0 or later; you must either drop the partitioning from such tables with ALTER TABLE ... REMOVE PARTITIONING or convert them to InnoDB using ALTER TABLE ... ENGINE=INNODB prior to the upgrade.

For information about converting MyISAM tables to InnoDB, see Section 15.6.1.5, “Converting Tables from MyISAM to InnoDB” .

**24.6.3** **Partitioning** **Limitations** **Relating** **to** **Functions**

This section discusses limitations in MySQL Partitioning relating specifically to functions used in partitioning expressions.

Only the MySQL functions shown in the following list are allowed in partitioning expressions:

• ABS()

• CEILING() (see [CEILING() and FLOOR()](#_bookmark21))

• DATEDIFF()

• DAY()

• DAYOFMONTH()

• DAYOFWEEK()

• DAYOFYEAR()

• EXTRACT() (see [EXTRACT() function with WEEK specifier](#_bookmark22))

• FLOOR() (see [CEILING() and FLOOR()](#_bookmark21))

• HOUR()

• MICROSECOND()

• MINUTE()

**PARTITION** **p0** **VALUES** **IN** **(1,3,5),**

**PARTITION** **p1** **VALUES** **IN** **(2,4,6)**

ERROR 1490

• MOD()

• MONTH()

• QUARTER()

• SECOND()

• TIME\_TO\_SEC()

• TO\_DAYS()

• TO\_SECONDS()

• UNIX\_TIMESTAMP() (with TIMESTAMP columns)

• WEEKDAY()

• YEAR()

• YEARWEEK()

In MySQL 8.0, partition pruning is supported for the TO\_DAYS(), TO\_SECONDS(), YEAR(), and UNIX\_TIMESTAMP() functions. See [Section 24.4, “Partition Pruning”](#_bookmark7) , for more information.

**CEILING()** **and** **FLOOR().** Each of these functions returns an integer only if it is passed an argument of an exact numeric type, such as one of the INT types or DECIMAL. This means, for example, that the following CREATE TABLE statement fails with an error, as shown here:

|  |  |
| --- | --- |
| mysql> **CREATE** **TABLE** **t** **(c** **FLOAT)** **PARTITION** **BY** **LIST(**  ->  ->  -> **);**  (HY000): The PARTITION function returns | **FLOOR(c)** **)(**  the wrong type |

**EXTRACT()** **function** **with** **WEEK** **specifier.** The value returned by the EXTRACT() function, when used as EXTRACT(WEEK FROM *col*), depends on the value of the default\_week\_format system variable. For this reason, EXTRACT() is not permitted as a partitioning function when it specifies the unit as WEEK. (Bug #54483)

See Section 12.6.2, “Mathematical Functions” , for more information about the return types of these functions, as well as Section 11.1, “Numeric Data Types” .

Chapter 25 Stored Objects

**Table** **of** **Contents**

[25.1 Defining Stored Programs](#_bookmark24) [4800](#_bookmark24)

[25.2 Using Stored Routines](#_bookmark25) [4801](#_bookmark25)

[25.2.1 Stored Routine Syntax](#_bookmark26) [4802](#_bookmark26)

[25.2.2 Stored Routines MySQL Privileges](#_bookmark27)and [4802](#_bookmark27)

[25.2.3 Stored Routine Metadata](#_bookmark28) [4803](#_bookmark28)

[25.2.4 Stored Procedures, Functions, Triggers, LAST\_INSERT\_ID()](#_bookmark29)and [4803](#_bookmark29)

[25.3 Using Triggers](#_bookmark30) [4803](#_bookmark30)

[25.3.1 Trigger Syntax Examples](#_bookmark31)and [4804](#_bookmark31)

[25.3.2 Trigger Metadata](#_bookmark32) [4808](#_bookmark32)

[25.4 Using Event Scheduler](#_bookmark33)the [4808](#_bookmark33)

[25.4.1 Event Scheduler Overview](#_bookmark34) [4809](#_bookmark34)

[25.4.2 Event Scheduler Configuration](#_bookmark35) [4809](#_bookmark35)

[25.4.3 Event Syntax](#_bookmark36) [4812](#_bookmark36)

[25.4.4 Event Metadata](#_bookmark37) [4812](#_bookmark37)

[25.4.5 Event Scheduler Status](#_bookmark38) [4813](#_bookmark38)

[25.4.6 The Event Scheduler MySQL Privileges](#_bookmark39)and [4813](#_bookmark39)

[25.5 Using Views](#_bookmark40) [4816](#_bookmark40)

[25.5.1 View Syntax](#_bookmark41) [4816](#_bookmark41)

[25.5.2 View Processing Algorithms](#_bookmark42) [4816](#_bookmark42)

[25.5.3 Updatable Insertable Views](#_bookmark43)and [4817](#_bookmark43)

[25.5.4 The View WITH CHECK OPTION Clause](#_bookmark44) [4820](#_bookmark44)

[25.5.5 View Metadata](#_bookmark45) [4821](#_bookmark45)

[25.6 Stored Object Access Control](#_bookmark46) [4821](#_bookmark46)

[25.7 Stored Program Binary Logging](#_bookmark47) [4825](#_bookmark47)

[25.8 Restrictions Stored Programs](#_bookmark48)on [4831](#_bookmark48)

[25.9 Restrictions Views](#_bookmark49)on [4834](#_bookmark49)

This chapter discusses stored database objects that are defined in terms of SQL code that is stored on the server for later execution.

Stored objects include these object types:

• Stored procedure: An object created with CREATE PROCEDURE and invoked using the CALL statement. A procedure does not have a return value but can modify its parameters for later inspection by the caller. It can also generate result sets to be returned to the client program.

• Stored function: An object created with CREATE FUNCTION and used much like a built-in function. You invoke it in an expression and it returns a value during expression evaluation.

• Trigger: An object created with CREATE TRIGGER that is associated with a table. A trigger is activated when a particular event occurs for the table, such as an insert or update.

• Event: An object created with CREATE EVENT and invoked by the server according to schedule.

• View: An object created with CREATE VIEW that when referenced produces a result set. A view acts as a virtual table.

Terminology used in this document reflects the stored object hierarchy:

• Stored routines include stored procedures and functions.

• Stored programs include stored routines, triggers, and events.

• Stored objects include stored programs and views.

This chapter describes how to use stored objects. The following sections provide additional information about SQL syntax for statements related to these objects, and about object processing:

• For each object type, there are CREATE, ALTER, and DROP statements that control which objects exist and how they are defined. See Section 13.1, “Data Definition Statements” .

• The CALL statement is used to invoke stored procedures. See Section 13.2.1, “CALL Statement” .

• Stored program definitions include a body that may use compound statements, loops, conditionals, and declared variables. See Section 13.6, “Compound Statement Syntax” .

• Metadata changes to objects referred to by stored programs are detected and cause automatic reparsing of the affected statements when the program is next executed. For more information, see Section 8.10.3, “Caching of Prepared Statements and Stored Programs” .

**25.1** **Defining** **Stored** **Programs**

Each stored program contains a body that consists of an SQL statement. This statement may be a compound statement made up of several statements separated by semicolon (;) characters. For example, the following stored procedure has a body made up of a BEGIN ... END block that contains a SET statement and a REPEAT loop that itself contains another SET statement:

CREATE PROCEDURE dorepeat(p1 INT)

BEGIN

SET @x = 0;

REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;

END;

If you use the mysql client program to define a stored program containing semicolon characters, a problem arises. By default, mysql itself recognizes the semicolon as a statement delimiter, so you must redefine the delimiter temporarily to cause mysql to pass the entire stored program definition to the server.

To redefine the mysql delimiter, use the delimiter command. The following example shows how to do this for the dorepeat() procedure just shown. The delimiter is changed to // to enable the entire definition to be passed to the server as a single statement, and then restored to ; before invoking the procedure. This enables the ; delimiter used in the procedure body to be passed through to the server rather than being interpreted by mysql itself.

mysql> **delimiter** **//**

mysql> **CREATE** **PROCEDURE** **dorepeat(p1** **INT)**

-> **BEGIN**

-> **SET** **@x** **=** **0;**

-> **REPEAT** **SET** **@x** **=** **@x** **+** **1;** **UNTIL** **@x** **>** **p1** **END** **REPEAT;**

-> **END**

-> **//**

Query OK, 0 rows affected (0.00 sec)

mysql> **delimiter** **;**

mysql> **CALL** **dorepeat(1000);**

Query OK, 0 rows affected (0.00 sec)

mysql> **SELECT** **@x;**

+------+

| @x |

+------+

| 1001 |

+------+

1 row in set (0.00 sec)

You can redefine the delimiter to a string other than //, and the delimiter can consist of a single character or multiple characters. You should avoid the use of the backslash (\) character because that is the escape character for MySQL.

The following is an example of a function that 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)

**25.2** **Using** **Stored** **Routines**

MySQL supports stored routines (procedures and functions). A stored routine is a set of SQL statements that can be stored in the server. Once this has been done, clients don't need to keep reissuing the individual statements but can refer to the stored routine instead.

Stored routines can be particularly useful in certain situations:

• When multiple client applications are written in different languages or work on different platforms, but need to perform the same database operations.

• When security is paramount. Banks, for example, use stored procedures and functions for all common operations. This provides a consistent and secure environment, and routines can ensure that each operation is properly logged. In such a setup, applications and users would have no access to the database tables directly, but can only execute specific stored routines.

Stored routines can provide improved performance because less information needs to be sent between the server and the client. The tradeoff is that this does increase the load on the database server because more of the work is done on the server side and less is done on the client (application)

side. Consider this if many client machines (such as Web servers) are serviced by only one or a few database servers.

Stored routines also enable you to have libraries of functions in the database server. This is a feature shared by modern application languages that enable such design internally (for example, by using classes). Using these client application language features is beneficial for the programmer even outside the scope of database use.

MySQL follows the SQL:2003 syntax for stored routines, which is also used by IBM's DB2. All syntax described here is supported and any limitations and extensions are documented where appropriate.

**Additional** **Resources**

• You may find the [Stored Procedures User Forum](https://forums.mysql.com/list.php?98) of use when working with stored procedures and functions.

• For answers to some commonly asked questions regarding stored routines in MySQL, see Section A.4, “MySQL 8.0 FAQ: Stored Procedures and Functions” .

• There are some restrictions on the use of stored routines. See [Section 25.8, “Restrictions on Stored](#_bookmark48) [Programs”](#_bookmark48) .

• Binary logging for stored routines takes place as described in [Section 25.7, “Stored Program Binary](#_bookmark47) [Logging”](#_bookmark47) .

**25.2.1** **Stored** **Routine** **Syntax**

A stored routine is either a procedure or a function. Stored routines are created with the CREATE PROCEDURE and CREATE FUNCTION statements (see Section 13.1.17, “CREATE PROCEDURE and CREATE FUNCTION Statements” ). A procedure is invoked using a CALL statement (see Section 13.2.1, “CALL Statement”), and can only pass back values using output variables. A function can be called from inside a statement just like any other function (that is, by invoking the function's name), and can return a scalar value. The body of a stored routine can use compound statements (see Section 13.6, “Compound Statement Syntax”).

Stored routines can be dropped with the DROP PROCEDURE and DROP FUNCTION statements (see Section 13.1.29, “DROP PROCEDURE and DROP FUNCTION Statements”), and altered with the ALTER PROCEDURE and ALTER FUNCTION statements (see Section 13.1.7, “ALTER PROCEDURE Statement” ).

A stored procedure or function is associated with a particular database. This has several implications:

• When the routine is invoked, an implicit USE *db\_name* is performed (and undone when the routine terminates). USE statements within stored routines are not permitted.

• You can qualify routine names with the database name. This can be used to refer to a routine that is not in the current database. For example, to invoke a stored procedure p or function f that is associated with the test database, you can say CALL test.p() or test.f().

• When a database is dropped, all stored routines associated with it are dropped as well. Stored functions cannot be recursive.

Recursion in stored procedures is permitted but disabled by default. To enable recursion, set the max\_sp\_recursion\_depth server system variable to a value greater than zero. Stored procedure recursion increases the demand on thread stack space. If you increase the value of max\_sp\_recursion\_depth, it may be necessary to increase thread stack size by increasing the value of thread\_stack at server startup. See Section 5.1.8, “Server System Variables” , for more information.

MySQL supports a very useful extension that enables the use of regular SELECT statements (that

is, without using cursors or local variables) inside a stored procedure. The result set of such a query is simply sent directly to the client. Multiple SELECT statements generate multiple result sets, so the client must use a MySQL client library that supports multiple result sets. This means the client must use a client library from a version of MySQL at least as recent as 4.1. The client should also specify the CLIENT\_MULTI\_RESULTS option when it connects. For C programs, this can be done with the [mysql\_real\_connect()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-real-connect.html) C API function. See [mysql\_real\_connect()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-real-connect.html), and [Multiple Statement](https://dev.mysql.com/doc/c-api/8.0/en/c-api-multiple-queries.html) [Execution Support](https://dev.mysql.com/doc/c-api/8.0/en/c-api-multiple-queries.html).

In MySQL 8.0.22 and later, a user variable referenced by a statement in a stored procedure has its type determined the first time the procedure is invoked, and retains this type each time the procedure is invoked thereafter.

**25.2.2** **Stored** **Routines** **and** **MySQL** **Privileges**

The MySQL grant system takes stored routines into account as follows:

• The CREATE ROUTINE privilege is needed to create stored routines.

• The ALTER ROUTINE privilege is needed to alter or drop stored routines. This privilege is granted automatically to the creator of a routine if necessary, and dropped from the creator when the routine is dropped.

• The EXECUTE privilege is required to execute stored routines. However, this privilege is granted automatically to the creator of a routine if necessary (and dropped from the creator when the routine is dropped). Also, the default SQL SECURITY characteristic for a routine is DEFINER, which enables users who have access to the database with which the routine is associated to execute the routine.

• If the automatic\_sp\_privileges system variable is 0, the EXECUTE and ALTER ROUTINE privileges are not automatically granted to and dropped from the routine creator.

• The creator of a routine is the account used to execute the CREATE statement for it. This might not be the same as the account named as the DEFINER in the routine definition.

• The account named as a routine DEFINER can see all routine properties, including its definition. The account thus has full access to the routine output as produced by:

• The contents of the Information Schema [ROUTINES](#_bookmark50) table.

• The SHOW CREATE FUNCTION and SHOW CREATE PROCEDURE statements.

• The SHOW FUNCTION CODE and SHOW PROCEDURE CODE statements.

• The SHOW FUNCTION STATUS and SHOW PROCEDURE STATUS statements.

• For an account other than the account named as the routine DEFINER, access to routine properties depends on the privileges granted to the account:

• With the SHOW\_ROUTINE privilege or the global SELECT privilege, the account can see all routine properties, including its definition.

• With the CREATE ROUTINE, ALTER ROUTINE or EXECUTE privilege granted at a scope that includes the routine, the account can see all routine properties except its definition.

**25.2.3** **Stored** **Routine** **Metadata**

To obtain metadata about stored routines:

• Query the [ROUTINES](#_bookmark50) table of the INFORMATION\_SCHEMA database. See [Section 26.3.30, “The](#_bookmark50) [INFORMATION\_SCHEMA ROUTINES Table”](#_bookmark50) .

• Use the SHOW CREATE PROCEDURE and SHOW CREATE FUNCTION statements to see routine definitions. See Section 13.7.7.9, “SHOW CREATE PROCEDURE Statement” .

• Use the SHOW PROCEDURE STATUS and SHOW FUNCTION STATUS statements to see routine characteristics. See Section 13.7.7.28, “SHOW PROCEDURE STATUS Statement” .

• Use the SHOW PROCEDURE CODE and SHOW FUNCTION CODE statements to see a representation of the internal implementation of the routine. See Section 13.7.7.27, “SHOW PROCEDURE CODE Statement” .

**25.2.4** **Stored** **Procedures,** **Functions,** **Triggers,** **and** **LAST\_INSERT\_ID()**

Within the body of a stored routine (procedure or function) or a trigger, the value of LAST\_INSERT\_ID() changes the same way as for statements executed outside the body of these kinds of objects (see Section 12.16, “Information Functions” ). The effect of a stored routine or trigger upon the value of LAST\_INSERT\_ID() that is seen by following statements depends on the kind of routine:

• If a stored procedure executes statements that change the value of LAST\_INSERT\_ID(), the changed value is seen by statements that follow the procedure call.

• For stored functions and triggers that change the value, the value is restored when the function or trigger ends, so following statements do not see a changed value.

**25.3** **Using** **Triggers**

A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table. Some uses for triggers are to perform checks of values to be inserted into a table or to perform calculations on values involved in an update.



A trigger is defined to activate when a statement inserts, updates, or deletes rows in the associated table. These row operations are trigger events. For example, rows can be inserted by INSERT or LOAD DATA statements, and an insert trigger activates for each inserted row. A trigger can be set to activate either before or after the trigger event. For example, you can have a trigger activate before each row that is inserted into a table or after each row that is updated.

**Important**

MySQL triggers activate only for changes made to tables by SQL statements. This includes changes to base tables that underlie updatable views. Triggers do not activate for changes to tables made by APIs that do not transmit SQL statements to the MySQL Server. This means that triggers are not activated by updates made using the NDB API.

Triggers are not activated by changes in INFORMATION\_SCHEMA or performance\_schema tables. Those tables are actually views and triggers are not permitted on views.

The following sections describe the syntax for creating and dropping triggers, show some examples of how to use them, and indicate how to obtain trigger metadata.

**Additional** **Resources**

• You may find the [MySQL User Forums](https://forums.mysql.com/list.php?20) helpful when working with triggers.

• For answers to commonly asked questions regarding triggers in MySQL, see Section A.5, “MySQL 8.0 FAQ: Triggers” .

• There are some restrictions on the use of triggers; see [Section 25.8, “Restrictions on Stored](#_bookmark48) [Programs”](#_bookmark48) .

• Binary logging for triggers takes place as described in [Section 25.7, “Stored Program Binary](#_bookmark47) [Logging”](#_bookmark47) .

**25.3.1** **Trigger** **Syntax** **and** **Examples**

To create a trigger or drop a trigger, use the CREATE TRIGGER or DROP TRIGGER statement, described in Section 13.1.22, “CREATE TRIGGER Statement” , and Section 13.1.34, “DROP TRIGGER

Statement” .

Here is a simple example that associates a trigger with a table, to activate for INSERT operations. The trigger acts as an accumulator, summing the values inserted into one of the columns of the table.

mysql> **CREATE** **TABLE** **account** **(acct\_num** **INT,** **amount** **DECIMAL(10,2));**

Query OK, 0 rows affected (0.03 sec)

mysql> **CREATE** **TRIGGER** **ins\_sum** **BEFORE** **INSERT** **ON** **account**

**FOR** **EACH** **ROW** **SET** **@sum** **=** **@sum** **+** **NEW** **.amount;**

Query OK, 0 rows affected (0.01 sec)

The CREATE TRIGGER statement creates a trigger named ins\_sum that is associated with the account table. It also includes clauses that specify the trigger action time, the triggering event, and what to do when the trigger activates:

• The keyword BEFORE indicates the trigger action time. In this case, the trigger activates before each row inserted into the table. The other permitted keyword here is AFTER.

• The keyword INSERT indicates the trigger event; that is, the type of operation that activates the trigger. In the example, INSERT operations cause trigger activation. You can also create triggers for DELETE and UPDATE operations.

• The statement following FOR EACH ROW defines the trigger body; that is, the statement to execute each time the trigger activates, which occurs once for each row affected by the triggering event.

In the example, the trigger body is a simple SET that accumulates into a user variable the values inserted into the amount column. The statement refers to the column as NEW.amount which means “the value of the amount column to be inserted into the new row.”

To use the trigger, set the accumulator variable to zero, execute an INSERT statement, and then see what value the variable has afterward:

mysql> **SET** **@sum** **=** **0;**

mysql> **INSERT** **INTO** **account** **VALUES(137,14** **.98),(141,1937** **.50),(97,-100** **.00);**

mysql> **SELECT** **@sum** **AS** **'Total** **amount** **inserted';**

+-----------------------+

| Total amount inserted |

+-----------------------+

| 1852.48 |

+-----------------------+

In this case, the value of @sum after the INSERT statement has executed is 14.98 + 1937.50 - 100, or 1852.48.

To destroy the trigger, use a DROP TRIGGER statement. You must specify the schema name if the trigger is not in the default schema:

mysql> **DROP** **TRIGGER** **test.ins\_sum;**

If you drop a table, any triggers for the table are also dropped.

Trigger names exist in the schema namespace, meaning that all triggers must have unique names within a schema. Triggers in different schemas can have the same name.

It is possible to define multiple triggers for a given table that have the same trigger event and action time. For example, you can have two BEFORE UPDATE triggers for a table. By default, triggers that have the same trigger event and action time activate in the order they were created. To affect trigger order, specify a clause after FOR EACH ROW that indicates FOLLOWS or PRECEDES and the name of an existing trigger that also has the same trigger event and action time. With FOLLOWS, the new trigger activates after the existing trigger. With PRECEDES, the new trigger activates before the existing trigger.

For example, the following trigger definition defines another BEFORE INSERT trigger for the account table:

mysql> **CREATE** **TRIGGER** **ins\_transaction** **BEFORE** **INSERT** **ON** **account**

**FOR** **EACH** **ROW** **PRECEDES** **ins\_sum**

**SET**

**@deposits** **=** **@deposits** **+** **IF(NEW** **.amount>0,NEW** **.amount,0),**

**@withdrawals** **=** **@withdrawals** **+** **IF(NEW** **.amount<0,-NEW** **.amount,0);**

Query OK, 0 rows affected (0.01 sec)

This trigger, ins\_transaction, is similar to ins\_sum but accumulates deposits and withdrawals separately. It has a PRECEDES clause that causes it to activate before ins\_sum; without that clause, it would activate after ins\_sum because it is created after ins\_sum.

Within the trigger body, the OLD and NEW keywords enable you to access columns in the rows affected by a trigger. OLD and NEW are MySQL extensions to triggers; they are not case-sensitive.

In an INSERT trigger, only NEW.*col\_name* can be used; there is no old row. In a DELETE trigger, only OLD.*col\_name* can be used; there is no new row. In an UPDATE trigger, you can use OLD.*col\_name* to refer to the columns of a row before it is updated and NEW.*col\_name* to refer to the columns of the row after it is updated.

A column named with OLD is read only. You can refer to it (if you have the SELECT privilege), but not modify it. You can refer to a column named with NEW if you have the SELECT privilege for it. In a BEFORE trigger, you can also change its value with SET NEW.*col\_name* = *value* if you have the UPDATE privilege for it. This means you can use a trigger to modify the values to be inserted into a new row or used to update a row. (Such a SET statement has no effect in an AFTER trigger because the row change has already occurred.)

In a BEFORE trigger, the NEW value for an AUTO\_INCREMENT column is 0, not the sequence number that is generated automatically when the new row actually is inserted.

By using the BEGIN ... END construct, you can define a trigger that executes multiple statements. Within the BEGIN block, you also can use other syntax that is permitted within stored routines such as conditionals and loops. However, just as for stored routines, if you use the mysql program to define a trigger that executes multiple statements, it is necessary to redefine the mysql statement delimiter so that you can use the ; statement delimiter within the trigger definition. The following example illustrates these points. It defines an UPDATE trigger that checks the new value to be used for updating each row, and modifies the value to be within the range from 0 to 100. This must be a BEFORE trigger because the value must be checked before it is used to update the row:

mysql> **delimiter** **//**

mysql> **CREATE** **TRIGGER** **upd\_check** **BEFORE** **UPDATE** **ON** **account**

**FOR** **EACH** **ROW**

**BEGIN**

**IF** **NEW.amount** **<** **0** **THEN**

**SET** **NEW.amount** **=** **0;**

**ELSEIF** **NEW.amount** **>** **100** **THEN**

**SET** **NEW.amount** **=** **100;**

**END** **IF;**

**END;//**

mysql> **delimiter** **;**

It can be easier to define a stored procedure separately and then invoke it from the trigger using a simple CALL statement. This is also advantageous if you want to execute the same code from within several triggers.

There are limitations on what can appear in statements that a trigger executes when activated:

• The trigger cannot use the CALL statement to invoke stored procedures that return data to the client or that use dynamic SQL. (Stored procedures are permitted to return data to the trigger through OUT or INOUT parameters.)

• The trigger cannot use statements that explicitly or implicitly begin or end a transaction, such as START TRANSACTION, COMMIT, or ROLLBACK. (ROLLBACK to SAVEPOINT is permitted because it does not end a transaction.).

See also [Section 25.8, “Restrictions on Stored Programs”](#_bookmark48) .

MySQL handles errors during trigger execution as follows:

• If a BEFORE trigger fails, the operation on the corresponding row is not performed.

• A BEFORE trigger is activated by the *attempt* to insert or modify the row, regardless of whether the attempt subsequently succeeds.

• An AFTER trigger is executed only if any BEFORE triggers and the row operation execute successfully.

• An error during either a BEFORE or AFTER trigger results in failure of the entire statement that caused trigger invocation.

• For transactional tables, failure of a statement should cause rollback of all changes performed by the statement. Failure of a trigger causes the statement to fail, so trigger failure also causes rollback. For nontransactional tables, such rollback cannot be done, so although the statement fails, any changes performed prior to the point of the error remain in effect.

Triggers can contain direct references to tables by name, such as the trigger named testref shown in this example:

|  |  |
| --- | --- |
| CREATE TABLE test1(a1  CREATE TABLE test2(a2  CREATE TABLE test3(a3 | INT);  INT);  INT NOT NULL AUTO\_INCREMENT PRIMARY KEY); |

CREATE TABLE test4(

a4 INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

b4 INT DEFAULT [0](#_bookmark51)

);

delimiter |

CREATE TRIGGER testref BEFORE INSERT ON test1

FOR EACH ROW

BEGIN

INSERT INTO test2 SET a2 = NEW.a1;

DELETE FROM test3 WHERE a3 = NEW .a1;

UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;

END;

|

delimiter ;

INSERT INTO test3 (a3) VALUES

(NULL), (NULL), (NULL), (NULL), (NULL),

(NULL), (NULL), (NULL), (NULL), (NULL);

INSERT INTO test4 (a4) VALUES

(0), (0), (0), (0), (0), (0), (0), (0), (0), (0);

Suppose that you insert the following values into table test1 as shown here:

mysql> **INSERT** **INTO** **test1** **VALUES**

**(1),** **(3),** **(1),** **(7),** **(1),** **(8),** **(4),** **(4);**

Query OK, 8 rows affected (0.01 sec)

Records: 8 Duplicates: 0 Warnings: [0](#_bookmark6)

As a result, the four tables contain the following data:

mysql> **SELECT** **\*** **FROM** **test1;**

+------+

| a1 |

+------+

|  |  |  |
| --- | --- | --- |
| |  |  |  |  |  |  |  | | 1  3  1  7  1  8  4  4 | |  |  |  |  |  |  |  | |

+------+

8 rows in set (0.00 sec)

mysql> **SELECT** **\*** **FROM** **test2;**

+------+

| a2 |

+------+

|  |  |  |
| --- | --- | --- |
| |  |  |  |  |  |  |  | | 1  3  1  7  1  8  4  4 | |  |  |  |  |  |  |  | |

+------+

8 rows in set (0.00 sec)

mysql> **SELECT** **\*** **FROM** **test3;**

+----+

| a3 |

+----+

|  |  |  |
| --- | --- | --- |
| |  |  |  | | 2  5  6  9 | |  |  |  | |

| 10 |

+----+

5 rows in set (0.00 sec)

mysql> **SELECT** **\*** **FROM** **test4;**

+----+------+

| a4 | b4 |

+----+------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  | | 1  2  3  4  5  6  7  8  9  10 | |  |  |  |  |  |  |  |  |  | | 3  0  1  2  0  0  1  1  0  0 | |  |  |  |  |  |  |  |  |  | |

+----+------+

10 rows in set (0.00 sec)

**25.3.2** **Trigger** **Metadata**

To obtain metadata about triggers:

• Query the [TRIGGERS](#_bookmark52) table of the INFORMATION\_SCHEMA database. See [Section 26.3.45, “The](#_bookmark52) [INFORMATION\_SCHEMA TRIGGERS Table”](#_bookmark52) .

• Use the SHOW CREATE TRIGGER statement. See Section 13.7.7.11, “SHOW CREATE TRIGGER Statement” .

• Use the SHOW TRIGGERS statement. See Section 13.7.7.40, “SHOW TRIGGERS Statement” .

**25.4** **Using** **the** **Event** **Scheduler**

The *MySQL* *Event* *Scheduler* manages the scheduling and execution of events, that is, tasks that run according to a schedule. The following discussion covers the Event Scheduler and is divided into the following sections:

• [Section 25.4.1, “Event Scheduler Overview”](#_bookmark34) , provides an introduction to and conceptual overview of MySQL Events.

• [Section 25.4.3, “Event Syntax”](#_bookmark36) , discusses the SQL statements for creating, altering, and dropping MySQL Events.

• [Section 25.4.4, “Event Metadata”](#_bookmark37) , shows how to obtain information about events and how this information is stored by the MySQL Server.

• [Section 25.4.6, “The Event Scheduler and MySQL Privileges”](#_bookmark39) , discusses the privileges required to work with events and the ramifications that events have with regard to privileges when executing.

Stored routines require the events data dictionary table in the mysql system database. This table is created during the MySQL 8.0 installation procedure. If you are upgrading to MySQL 8.0 from an earlier version, be sure to perform the upgrade procedure to make sure that your system database is up to date. See Section 2.10, “Upgrading MySQL” .

**Additional** **Resources**

• There are some restrictions on the use of events; see [Section 25.8, “Restrictions on Stored](#_bookmark48) [Programs”](#_bookmark48) .

• Binary logging for events takes place as described in [Section 25.7, “Stored Program Binary Logging”](#_bookmark47) .

• You may also find the [MySQL User Forums](https://forums.mysql.com/list.php?20) to be helpful.

**25.4.1** **Event** **Scheduler** **Overview**

MySQL Events are tasks that run according to a schedule. Therefore, we sometimes refer to them as *scheduled* events. When you create an event, you are creating a named database object containing one or more SQL statements to be executed at one or more regular intervals, beginning and ending at a specific date and time. Conceptually, this is similar to the idea of the Unix crontab (also known as a “cron job”) or the Windows Task Scheduler.

Scheduled tasks of this type are also sometimes known as “temporal triggers” , implying that these are objects that are triggered by the passage of time. While this is essentially correct, we prefer to use the term *events* to avoid confusion with triggers of the type discussed in [Section 25.3, “Using Triggers”](#_bookmark30) . Events should more specifically not be confused with “temporary triggers” . Whereas a trigger is a database object whose statements are executed in response to a specific type of event that occurs on a given table, a (scheduled) event is an object whose statements are executed in response to the passage of a specified time interval.

While there is no provision in the SQL Standard for event scheduling, there are precedents in other database systems, and you may notice some similarities between these implementations and that found in the MySQL Server.

MySQL Events have the following major features and properties:

• In MySQL, an event is uniquely identified by its name and the schema to which it is assigned.

• An event performs a specific action according to a schedule. This action consists of an SQL statement, which can be a compound statement in a BEGIN ... END block if desired (see Section 13.6, “Compound Statement Syntax”). An event's timing can be either *one-time* or *recurrent*. A one-time event executes one time only. A recurrent event repeats its action at a regular interval, and the schedule for a recurring event can be assigned a specific start day and time, end day and time, both, or neither. (By default, a recurring event's schedule begins as soon as it is created, and continues indefinitely, until it is disabled or dropped.)

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.

• Users can create, modify, and drop scheduled events using SQL statements intended for these purposes. Syntactically invalid event creation and modification statements fail with an appropriate error message. *A* *user* *may* *include* *statements* *in* *an* *event's* *action* *which* *require* *privileges* *that* *the* *user* *does* *not* *actually* *have*. The event creation or modification statement succeeds but the event's action fails. See [Section 25.4.6, “The Event Scheduler and MySQL Privileges”](#_bookmark39) for details.

• Many of the properties of an event can be set or modified using SQL statements. These properties include the event's name, timing, persistence (that is, whether it is preserved following the expiration of its schedule), status (enabled or disabled), action to be performed, and the schema to which it is assigned. See Section 13.1.3, “ALTER EVENT Statement” .

The default definer of an event is the user who created the event, unless the event has been altered, in which case the definer is the user who issued the last ALTER EVENT statement affecting that event. An event can be modified by any user having the EVENT privilege on the database for which the event is defined. See [Section 25.4.6, “The Event Scheduler and MySQL Privileges”](#_bookmark39) .

• An event's action statement may include most SQL statements permitted within stored routines. For restrictions, see [Section 25.8, “Restrictions on Stored Programs”](#_bookmark48) .

**25.4.2** **Event** **Scheduler** **Configuration**

Events are executed by a special *event* *scheduler* *thread*; when we refer to the Event Scheduler, we actually refer to this thread. When running, the event scheduler thread and its current state can be



seen by users having the PROCESS privilege in the output of SHOW PROCESSLIST, as shown in the discussion that follows.

The global event\_scheduler system variable determines whether the Event Scheduler is enabled and running on the server. It has one of the following values, which affect event scheduling as described:

• ON: The Event Scheduler is started; the event scheduler thread runs and executes all scheduled events. ON is the default event\_scheduler value.

When the Event Scheduler is ON, the event scheduler thread is listed in the output of SHOW PROCESSLIST as a daemon process, and its state is represented as shown here:

mysql> **SHOW** **PROCESSLIST\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Id: 1

User: root

Host: localhost

db: NULL

Command: Query

Time: 0

State: NULL

Info: show processlist

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Id: 2

User: event\_scheduler

Host: localhost

db: NULL

Command: Daemon

Time: 3

State: Waiting for next activation

Info: NULL

2 rows in set (0.00 sec)

Event scheduling can be stopped by setting the value of event\_scheduler to OFF.

• OFF: The Event Scheduler is stopped. The event scheduler thread does not run, is not shown in the output of SHOW PROCESSLIST, and no scheduled events execute.

When the Event Scheduler is stopped (event\_scheduler is OFF), it can be started by setting the value of event\_scheduler to ON. (See next item.)

• DISABLED: This value renders the Event Scheduler nonoperational. When the Event Scheduler is DISABLED, the event scheduler thread does not run (and so does not appear in the output of SHOW PROCESSLIST). In addition, the Event Scheduler state cannot be changed at runtime.

If the Event Scheduler status has not been set to DISABLED, event\_scheduler can be toggled between ON and OFF (using SET). It is also possible to use 0 for OFF, and 1 for ON when setting this variable. Thus, any of the following 4 statements can be used in the mysql client to turn on the Event

Scheduler:

SET GLOBAL event\_scheduler = ON;

SET @@GLOBAL.event\_scheduler = ON;

SET GLOBAL event\_scheduler = 1;

SET @@GLOBAL.event\_scheduler = 1;

Similarly, any of these 4 statements can be used to turn off the Event Scheduler:

SET GLOBAL event\_scheduler = OFF;

SET @@GLOBAL.event\_scheduler = OFF;

SET GLOBAL event\_scheduler = 0;

SET @@GLOBAL.event\_scheduler = 0;

**Note**

If the Event Scheduler is enabled, enabling the super\_read\_only system variable prevents it from updating event “last executed” timestamps in the



events data dictionary table. This causes the Event Scheduler to stop the next time it tries to execute a scheduled event, after writing a message to the server error log. (In this situation the event\_scheduler system variable does not change from ON to OFF. An implication is that this variable rejects the DBA *intent* that the Event Scheduler be enabled or disabled, where its actual status of started or stopped may be distinct.). If super\_read\_only is subsequently disabled after being enabled, the server automatically restarts the Event Scheduler as needed, as of MySQL 8.0.26. Prior to MySQL 8.0.26, it is necessary to manually restart the Event Scheduler by enabling it again.

Although ON and OFF have numeric equivalents, the value displayed for event\_scheduler by SELECT or SHOW VARIABLES is always one of OFF, ON, or DISABLED. *D工SABLED* *has* *no* *numeric* *equivalent*. For this reason, ON and OFF are usually preferred over 1 and 0 when setting this variable.

Note that attempting to set event\_scheduler without specifying it as a global variable causes an error:

mysql< **SET** **@@event\_scheduler** **=** **OFF;**

ERROR 1229 (HY000): Variable 'event\_scheduler' is a GLOBAL

variable and should be set with SET GLOBAL

**Important**

It is possible to set the Event Scheduler to DISABLED only at server startup. If event\_scheduler is ON or OFF, you cannot set it to DISABLED at runtime. Also, if the Event Scheduler is set to DISABLED at startup, you cannot change the value of event\_scheduler at runtime.

To disable the event scheduler, use one of the following two methods:

• As a command-line option when starting the server: --event-scheduler=DISABLED

• In the server configuration file (my.cnf, or my.ini on Windows systems), include the line where it can be read by the server (for example, in a [mysqld] section):

event\_scheduler=DISABLED

To enable the Event Scheduler, restart the server without the --event-scheduler=DISABLED command-line option, or after removing or commenting out the line containing event- scheduler=DISABLED in the server configuration file, as appropriate. Alternatively, you can use ON (or 1) or OFF (or 0) in place of the DISABLED value when starting the server.

**Note**

You can issue event-manipulation statements when event\_scheduler is set to DISABLED. No warnings or errors are generated in such cases (provided that the statements are themselves valid). However, scheduled events cannot execute until this variable is set to ON (or 1). Once this has been done, the event scheduler thread executes all events whose scheduling conditions are satisfied.

Starting the MySQL server with the --skip-grant-tables option causes event\_scheduler to be set to DISABLED, overriding any other value set either on the command line or in the my.cnf or my.ini file (Bug #26807).

For SQL statements used to create, alter, and drop events, see [Section 25.4.3, “Event Syntax”](#_bookmark36) .

MySQL provides an [EVENTS](#_bookmark53) table in the INFORMATION\_SCHEMA database. This table can be queried to obtain information about scheduled events which have been defined on the server. See [Section 25.4.4, “Event Metadata”](#_bookmark37) , and [Section 26.3.14, “The INFORMATION\_SCHEMA EVENTS](#_bookmark53) [Table”](#_bookmark53) , for more information.

For information regarding event scheduling and the MySQL privilege system, see [Section 25.4.6, “The](#_bookmark39) [Event Scheduler and MySQL Privileges”](#_bookmark39) .

**25.4.3** **Event** **Syntax**

MySQL provides several SQL statements for working with scheduled events:

• New events are defined using the CREATE EVENT statement. See Section 13.1.13, “CREATE EVENT Statement” .

• The definition of an existing event can be changed by means of the ALTER EVENT statement. See Section 13.1.3, “ALTER EVENT Statement” .

• When a scheduled event is no longer wanted or needed, it can be deleted from the server by its definer using the DROP EVENT statement. See Section 13. 1.25, “DROP EVENT Statement” . Whether an event persists past the end of its schedule also depends on its ON COMPLETION clause,

if it has one. See Section 13. 1. 13, “CREATE EVENT Statement” .

An event can be dropped by any user having the EVENT privilege for the database on which the event is defined. See [Section 25.4.6, “The Event Scheduler and MySQL Privileges”](#_bookmark39) .

**25.4.4** **Event** **Metadata**

To obtain metadata about events:

• Query the [EVENTS](#_bookmark53) table of the INFORMATION\_SCHEMA database. See [Section 26.3.14, “The](#_bookmark53) [INFORMATION\_SCHEMA EVENTS Table”](#_bookmark53) .

• Use the SHOW CREATE EVENT statement. See Section 13.7.7.7, “SHOW CREATE EVENT Statement” .

• Use the SHOW EVENTS statement. See Section 13.7.7.18, “SHOW EVENTS Statement” .

**Event** **Scheduler** **Time** **Representation**

Each session in MySQL has a session time zone (STZ). This is the session time\_zone value that is initialized from the server's global time\_zone value when the session begins but may be changed during the session.

The session time zone that is current when a CREATE EVENT or ALTER EVENT statement executes is used to interpret times specified in the event definition. This becomes the event time zone (ETZ); that is, the time zone that is used for event scheduling and is in effect within the event as it executes.

For representation of event information in the data dictionary, the execute\_at, starts, and ends times are converted to UTC and stored along with the event time zone. This enables event execution to proceed as defined regardless of any subsequent changes to the server time zone or daylight saving time effects. The last\_executed time is also stored in UTC.

Event times can be obtained by selecting from the Information Schema [EVENTS](#_bookmark53) table or from SHOW EVENTS, but they are reported as ETZ or STZ values. The following table summarizes representation of event times.

|  |  |  |
| --- | --- | --- |
| **Value** | [**EVENTS**](#_bookmark53) **Table** | **SHOW** **EVENTS** |
| Execute at | ETZ | ETZ |
| Starts | ETZ | ETZ |
| Ends | ETZ | ETZ |
| Last executed | ETZ | n/a |
| Created | STZ | n/a |
| Last altered | STZ | n/a |

**25.4.5** **Event** **Scheduler** **Status**

The Event Scheduler writes information about event execution that terminates with an error or warning to the MySQL Server's error log. See [Section 25.4.6, “The Event Scheduler and MySQL Privileges”](#_bookmark39) for an example.

To obtain information about the state of the Event Scheduler for debugging and troubleshooting purposes, run mysqladmin debug (see Section 4.5.2, “mysqladmin — A MySQL Server Administration Program”); after running this command, the server's error log contains output relating to the Event Scheduler, similar to what is shown here:

Events status:

LLA = Last Locked At LUA = Last Unlocked At

WOC = Waiting On Condition DL = Data Locked

Event scheduler status:

State : INITIALIZED

Thread id : 0

LLA : n/a:0

LUA : n/a:0

WOC : NO

Workers : 0

Executed : 0

Data locked: NO

Event queue status:

Element count : 0

Data locked : NO

Attempting lock : NO

LLA : init\_queue:95

LUA : init\_queue:103

WOC : NO

Next activation : never

In statements that occur as part of events executed by the Event Scheduler, diagnostics messages (not only errors, but also warnings) are written to the error log, and, on Windows, to the application event log. For frequently executed events, it is possible for this to result in many logged messages. For example, for SELECT ... INTO *var\_list* statements, if the query returns no rows, a warning with error code 1329 occurs (No data), and the variable values remain unchanged. If the query returns multiple rows, error 1172 occurs (Result consisted of more than one row). For either condition, you can avoid having the warnings be logged by declaring a condition handler; see Section 13.6.7.2, “DECLARE ... HANDLER Statement” . For statements that may retrieve multiple rows, another strategy is to use LIMIT 1 to limit the result set to a single row.

**25.4.6** **The** **Event** **Scheduler** **and** **MySQL** **Privileges**

To enable or disable the execution of scheduled events, it is necessary to set the value of the global event\_scheduler system variable. This requires privileges sufficient to set global system variables. See Section 5.1.9.1, “System Variable Privileges” .

The EVENT privilege governs the creation, modification, and deletion of events. This privilege can be bestowed using GRANT. For example, this GRANT statement confers the EVENT privilege for the schema named myschema on the user jon@ghidora:

GRANT EVENT ON myschema.\* TO jon@ghidora;

(We assume that this user account already exists, and that we wish for it to remain unchanged otherwise.)

To grant this same user the EVENT privilege on all schemas, use the following statement:

GRANT EVENT ON \*.\* TO jon@ghidora;

The EVENT privilege has global or schema-level scope. Therefore, trying to grant it on a single table results in an error as shown:

mysql> **GRANT** **EVENT** **ON** **myschema** **.mytable** **TO** **jon@ghidora;**

ERROR 1144 (42000): Illegal GRANT/REVOKE command; please

consult the manual to see which privileges can be used

It is important to understand that an event is executed with the privileges of its definer, and that it cannot perform any actions for which its definer does not have the requisite privileges. For example, suppose that jon@ghidora has the EVENT privilege for myschema. Suppose also that this user has the SELECT privilege for myschema, but no other privileges for this schema. It is possible for jon@ghidora to create a new event such as this one:

CREATE EVENT e\_store\_ts

ON SCHEDULE

EVERY 10 SECOND

DO

INSERT INTO myschema.mytable VALUES (UNIX\_TIMESTAMP());

The user waits for a minute or so, and then performs a SELECT \* FROM mytable; query, expecting to see several new rows in the table. Instead, the table is empty. Since the user does not have the INSERT privilege for the table in question, the event has no effect.

If you inspect the MySQL error log (*hostname*.err), you can see that the event is executing, but the action it is attempting to perform fails:

2013-09-24T12:41:31.261992Z 25 [ERROR] Event Scheduler:

[jon@ghidora][cookbook.e\_store\_ts] INSERT command denied to user

'jon'@'ghidora' for table 'mytable'

2013-09-24T12:41:31.262022Z 25 [Note] Event Scheduler:

[jon@ghidora]. [myschema.e\_store\_ts] event execution failed.

2013-09-24T12:41:41.271796Z 26 [ERROR] Event Scheduler:

[jon@ghidora][cookbook.e\_store\_ts] INSERT command denied to user

'jon'@'ghidora' for table 'mytable'

2013-09-24T12:41:41.272761Z 26 [Note] Event Scheduler:

[jon@ghidora]. [myschema.e\_store\_ts] event execution failed.

Since this user very likely does not have access to the error log, it is possible to verify whether the event's action statement is valid by executing it directly:

mysql> **INSERT** **INTO** **myschema** **.mytable** **VALUES** **(UNIX\_TIMESTAMP());**

ERROR 1142 (42000): INSERT command denied to user

'jon'@'ghidora' for table 'mytable'

Inspection of the Information Schema [EVENTS](#_bookmark53) table shows that e\_store\_ts exists and is enabled, but its LAST\_EXECUTED column is NULL:

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.EVENTS**

>

>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

EVENT\_CATALOG: NULL

EVENT\_SCHEMA: myschema

EVENT\_NAME: e\_store\_ts

DEFINER: jon@ghidora

EVENT\_BODY: SQL

EVENT\_DEFINITION: INSERT INTO myschema.mytable VALUES (UNIX\_TIMESTAMP())

EVENT\_TYPE: RECURRING

EXECUTE\_AT: NULL

INTERVAL\_VALUE: 5

INTERVAL\_FIELD: SECOND

SQL\_MODE: NULL

STARTS: 0000-00-00 00:00:00

ENDS: 0000-00-00 00:00:00

STATUS: ENABLED

ON\_COMPLETION: NOT PRESERVE

CREATED: 2006-02-09 22:36:06

LAST\_ALTERED: 2006-02-09 22:36:06

LAST\_EXECUTED: NULL

EVENT\_COMMENT:

1 row in set (0.00 sec)

**WHERE** **EVENT\_NAME='e\_store\_ts'**

**AND** **EVENT\_SCHEMA='myschema'\G**



To rescind the EVENT privilege, use the REVOKE statement. In this example, the EVENT privilege on the schema myschema is removed from the jon@ghidora user account:

REVOKE EVENT ON myschema.\* FROM jon@ghidora;

**Important**

Revoking the EVENT privilege from a user does not delete or disable any events that may have been created by that user.

An event is not migrated or dropped as a result of renaming or dropping the user who created it.

Suppose that the user jon@ghidora has been granted the EVENT and INSERT privileges on the myschema schema. This user then creates the following event:

CREATE EVENT e\_insert

ON SCHEDULE

EVERY 7 SECOND

DO

INSERT INTO myschema.mytable;

After this event has been created, root revokes the EVENT privilege for jon@ghidora. However, e\_insert continues to execute, inserting a new row into mytable each seven seconds. The same would be true if root had issued either of these statements:

• DROP USER jon@ghidora;

• RENAME USER jon@ghidora TO someotherguy@ghidora;

You can verify that this is true by examining the Information Schema [EVENTS](#_bookmark53) table before and after issuing a DROP USER or RENAME USER statement.

Event definitions are stored in the data dictionary. To drop an event created by another user account, you must be the MySQL root user or another user with the necessary privileges.

Users' EVENT privileges are stored in the Event\_priv columns of the mysql.user and mysql.db tables. In both cases, this column holds one of the values 'Y' or 'N'. 'N' is the default.

mysql.user.Event\_priv is set to 'Y' for a given user only if that user has the global EVENT privilege (that is, if the privilege was bestowed using GRANT EVENT ON \*.\*). For a schema-level EVENT privilege, GRANT creates a row in mysql.db and sets that row's Db column to the name of the schema, the User column to the name of the user, and the Event\_priv column to 'Y'. There should never be any need to manipulate these tables directly, since the GRANT EVENT and REVOKE EVENT statements perform the required operations on them.

Five status variables provide counts of event-related operations (but *not* of statements executed by events; see [Section 25.8, “Restrictions on Stored Programs”](#_bookmark48)). These are:

• Com\_create\_event: The number of CREATE EVENT statements executed since the last server restart.

• Com\_alter\_event: The number of ALTER EVENT statements executed since the last server restart.

• Com\_drop\_event: The number of DROP EVENT statements executed since the last server restart.

• Com\_show\_create\_event: The number of SHOW CREATE EVENT statements executed since the last server restart.

• Com\_show\_events: The number of SHOW EVENTS statements executed since the last server restart.

You can view current values for all of these at one time by running the statement SHOW STATUS LIKE '%event%';.

**25.5** **Using** **Views**

MySQL supports views, including updatable views. Views are stored queries that when invoked produce a result set. A view acts as a virtual table.

The following discussion describes the syntax for creating and dropping views, and shows some examples of how to use them.

**Additional** **Resources**

• You may find the [MySQL User Forums](https://forums.mysql.com/list.php?20) helpful when working with views.

• For answers to some commonly asked questions regarding views in MySQL, see Section A.6, “MySQL 8.0 FAQ: Views” .

• There are some restrictions on the use of views; see [Section 25.9, “Restrictions on Views”](#_bookmark49) .

**25.5.1** **View** **Syntax**

The CREATE VIEW statement creates a new view (see Section 13.1.23, “CREATE VIEW Statement” ). To alter the definition of a view or drop a view, use ALTER VIEW (see Section 13.1.11, “ALTER VIEW Statement”), or DROP VIEW (see Section 13.1.35, “DROP VIEW Statement” ).

A view can be created from many kinds of SELECT statements. It can refer to base tables or other views. It can use joins, UNION, and subqueries. The SELECT need not even refer to any tables. The following example defines a view that selects two columns from another table, as well as an expression calculated from those columns:

mysql> **CREATE** **TABLE** **t** **(qty** **INT,** **price** **INT);**

mysql> **INSERT** **INTO** **t** **VALUES(3,** **50),** **(5,** **60);**

mysql> **CREATE** **VIEW** **v** **AS** **SELECT** **qty,** **price,** **qty\*price** **AS** **value** **FROM** **t;**

mysql> **SELECT** **\*** **FROM** **v;**

+------+-------+-------+

| qty | price | value |

+------+-------+-------+

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  | | 3  5 | |  | | 50  60 | |  | | 150  300 | |  | |

+------+-------+-------+

mysql> **SELECT** **\*** **FROM** **v** **WHERE** **qty** **=** **5;**

+------+-------+-------+

| qty | price | value |

+------+-------+-------+

| 5 | 60 | 300 |

+------+-------+-------+

**25.5.2** **View** **Processing** **Algorithms**

The optional ALGORITHM clause for CREATE VIEW or ALTER VIEW is a MySQL extension to standard SQL. It affects how MySQL processes the view. ALGORITHM takes three values: MERGE, TEMPTABLE, or UNDEFINED.

• For MERGE, the text of a statement that refers to the view and the view definition are merged such that parts of the view definition replace corresponding parts of the statement.

• For TEMPTABLE, the results from the view are retrieved into a temporary table, which then is used to execute the statement.

• For UNDEFINED, MySQL chooses which algorithm to use. It prefers MERGE over TEMPTABLE if possible, because MERGE is usually more efficient and because a view cannot be updatable if a temporary table is used.

• If no ALGORITHM clause is present, the default algorithm is determined by the value of the derived\_merge flag of the optimizer\_switch system variable. For additional discussion, see

Section 8.2.2.4, “Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization” .

A reason to specify TEMPTABLE explicitly is that locks can be released on underlying tables after the temporary table has been created and before it is used to finish processing the statement. This might result in quicker lock release than the MERGE algorithm so that other clients that use the view are not blocked as long.

A view algorithm can be UNDEFINED for three reasons:

• No ALGORITHM clause is present in the CREATE VIEW statement.

• The CREATE VIEW statement has an explicit ALGORITHM = UNDEFINED clause.

• ALGORITHM = MERGE is specified for a view that can be processed only with a temporary table. In this case, MySQL generates a warning and sets the algorithm to UNDEFINED.

As mentioned earlier, MERGE is handled by merging corresponding parts of a view definition into the statement that refers to the view. The following examples briefly illustrate how the MERGE algorithm works. The examples assume that there is a view v\_merge that has this definition:

CREATE ALGORITHM = MERGE VIEW v\_merge (vc1, vc2) AS

SELECT c1, c2 FROM t WHERE c3 > 100;

Example 1: Suppose that we issue this statement:

SELECT \* FROM v\_merge;

MySQL handles the statement as follows:

• v\_merge becomes t

• \* becomes vc1, vc2, which corresponds to c1, c2

• The view WHERE clause is added

The resulting statement to be executed becomes:

SELECT c1, c2 FROM t WHERE c3 > 100;

Example 2: Suppose that we issue this statement:

SELECT \* FROM v\_merge WHERE vc1 < 100;

This statement is handled similarly to the previous one, except that vc1 < 100 becomes c1 < 100 and the view WHERE clause is added to the statement WHERE clause using an AND connective (and parentheses are added to make sure the parts of the clause are executed with correct precedence). The resulting statement to be executed becomes:

SELECT c1, c2 FROM t WHERE (c3 > 100) AND (c1 < 100);

Effectively, the statement to be executed has a WHERE clause of this form:

WHERE (select WHERE) AND (view WHERE)

If the MERGE algorithm cannot be used, a temporary table must be used instead. Constructs that prevent merging are the same as those that prevent merging in derived tables and common table expressions. Examples are SELECT DISTINCT or LIMIT in the subquery. For details, see

Section 8.2.2.4, “Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization” .

**25.5.3** **Updatable** **and** **Insertable** **Views**

Some views are updatable and references to them can be used to specify tables to be updated in data change statements. That is, you can use them in statements such as UPDATE, DELETE, or INSERT to update the contents of the underlying table. Derived tables and common table expressions can also be

specified in multiple-table UPDATE and DELETE statements, but can only be used for reading data to specify rows to be updated or deleted. Generally, the view references must be updatable, meaning that they may be merged and not materialized. Composite views have more complex rules.

For a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table. There are also certain other constructs that make a view nonupdatable. To be more specific, a view is not updatable if it contains any of the following:

• Aggregate functions or window functions (SUM(), MIN(), MAX(), COUNT(), and so forth)

• DISTINCT

• GROUP BY

• HAVING

• UNION or UNION ALL

• Subquery in the select list

Nondependent subqueries in the select list fail for INSERT, but are okay for UPDATE, DELETE. For dependent subqueries in the select list, no data change statements are permitted.

• Certain joins (see additional join discussion later in this section)

• Reference to nonupdatable view in the FROM clause

• Subquery in the WHERE clause that refers to a table in the FROM clause

• Refers only to literal values (in this case, there is no underlying table to update)

• ALGORITHM = TEMPTABLE (use of a temporary table always makes a view nonupdatable)

• Multiple references to any column of a base table (fails for INSERT, okay for UPDATE, DELETE)

A generated column in a view is considered updatable because it is possible to assign to it. However, if such a column is updated explicitly, the only permitted value is DEFAULT. For information about generated columns, see Section 13.1.20.8, “CREATE TABLE and Generated Columns” .

It is sometimes possible for a multiple-table view to be updatable, assuming that it can be processed with the MERGE algorithm. For this to work, the view must use an inner join (not an outer join or a UNION). Also, only a single table in the view definition can be updated, so the SET clause must name only columns from one of the tables in the view. Views that use UNION ALL are not permitted even though they might be theoretically updatable.

With respect to insertability (being updatable with INSERT statements), an updatable view is insertable if it also satisfies these additional requirements for the view columns:

• There must be no duplicate view column names.

• The view must contain all columns in the base table that do not have a default value.

• The view columns must be simple column references. They must not be expressions, such as these:

3.14159

col1 + 3

UPPER(col2)

col3 / col4

(*subquery*)

MySQL sets a flag, called the view updatability flag, at CREATE VIEW time. The flag is set to YES (true) if UPDATE and DELETE (and similar operations) are legal for the view. Otherwise, the flag is set to NO (false). The IS\_UPDATABLE column in the Information Schema [VIEWS](#_bookmark54) table displays the status of this flag. It means that the server always knows whether a view is updatable.

If a view is not updatable, statements such UPDATE, DELETE, and INSERT are illegal and are rejected. (Even if a view is updatable, it might not be possible to insert into it, as described elsewhere in this section.)

The updatability of views may be affected by the value of the updatable\_views\_with\_limit system variable. See Section 5.1.8, “Server System Variables” .

For the following discussion, suppose that these tables and views exist:

CREATE TABLE t1 (x INTEGER);

CREATE TABLE t2 (c INTEGER);

CREATE VIEW vmat AS SELECT SUM(x) AS s FROM t1;

CREATE VIEW vup AS SELECT \* FROM t2;

CREATE VIEW vjoin AS SELECT \* FROM vmat JOIN vup ON vmat.s=vup.c;

INSERT, UPDATE, and DELETE statements are permitted as follows:

• INSERT: The insert table of an INSERT statement may be a view reference that is merged. If the view is a join view, all components of the view must be updatable (not materialized). For a multiple- table updatable view, INSERT can work if it inserts into a single table.

This statement is invalid because one component of the join view is nonupdatable: INSERT INTO vjoin (c) VALUES (1); This statement is valid; the view contains no materialized components: INSERT INTO vup (c) VALUES (1);

• UPDATE: The table or tables to be updated in an UPDATE statement may be view references that are merged. If a view is a join view, at least one component of the view must be updatable (this differs from INSERT).

In a multiple-table UPDATE statement, the updated table references of the statement must be base tables or updatable view references. Nonupdated table references may be materialized views or derived tables.

This statement is valid; column c is from the updatable part of the join view: UPDATE vjoin SET c=c+1; This statement is invalid; column x is from the nonupdatable part: UPDATE vjoin SET x=x+1;

This statement is valid; the updated table reference of the multiple-table UPDATE is an updatable view (vup):

UPDATE vup JOIN (SELECT SUM(x) AS s FROM t1) AS dt ON ...

SET c=c+1;

This statement is invalid; it tries to update a materialized derived table:

UPDATE vup JOIN (SELECT SUM(x) AS s FROM t1) AS dt ON ...

SET s=s+1;

• DELETE: The table or tables to be deleted from in a DELETE statement must be merged views. Join views are not allowed (this differs from INSERT and UPDATE).

This statement is invalid because the view is a join view: DELETE vjoin WHERE ...; This statement is valid because the view is a merged (updatable) view: DELETE vup WHERE ...;

This statement is valid because it deletes from a merged (updatable) view:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DELETE | vup | FROM | vup JOIN (SELECT | SUM(x) AS | s | FROM | t1) AS | dt ON ...; |

Additional discussion and examples follow.

Earlier discussion in this section pointed out that a view is not insertable if not all columns are simple column references (for example, if it contains columns that are expressions or composite expressions). Although such a view is not insertable, it can be updatable if you update only columns that are not expressions. Consider this view:

CREATE VIEW v AS SELECT col1, 1 AS col2 FROM t;

This view is not insertable because col2 is an expression. But it is updatable if the update does not try to update col2. This update is permissible:

UPDATE v SET col1 = 0;

This update is not permissible because it attempts to update an expression column:

UPDATE v SET col2 = 0;

If a table contains an AUTO\_INCREMENT column, inserting into an insertable view on the table that does not include the AUTO\_INCREMENT column does not change the value of LAST\_INSERT\_ID(), because the side effects of inserting default values into columns not part of the view should not be visible.

**25.5.4** **The** **View** **WITH** **CHECK** **OPTION** **Clause**

The WITH CHECK OPTION clause can be given for an updatable view to prevent inserts to rows for which the WHERE clause in the *select\_statement* is not true. It also prevents updates to rows for which the WHERE clause is true but the update would cause it to be not true (in other words, it prevents visible rows from being updated to nonvisible rows).

In a WITH CHECK OPTION clause for an updatable view, the LOCAL and CASCADED keywords determine the scope of check testing when the view is defined in terms of another view. When neither keyword is given, the default is CASCADED.

WITH CHECK OPTION testing is standard-compliant:

• With LOCAL, the view WHERE clause is checked, then checking recurses to underlying views and applies the same rules.

• With CASCADED, the view WHERE clause is checked, then checking recurses to underlying views, adds WITH CASCADED CHECK OPTION to them (for purposes of the check; their definitions remain unchanged), and applies the same rules.

• With no check option, the view WHERE clause is not checked, then checking recurses to underlying views, and applies the same rules.

Consider the definitions for the following table and set of views:

CREATE TABLE t1 (a INT);

CREATE VIEW v1 AS SELECT \* FROM t1 WHERE a < 2

WITH CHECK OPTION;

CREATE VIEW v2 AS SELECT \* FROM v1 WHERE a > [0](#_bookmark1)

WITH LOCAL CHECK OPTION;

CREATE VIEW v3 AS SELECT \* FROM v1 WHERE a > [0](#_bookmark4)

WITH CASCADED CHECK OPTION;

Here the v2 and v3 views are defined in terms of another view, v1.

Inserts for v2 are checked against its LOCAL check option, then the check recurses to v1 and the rules are applied again. The rules for v1 cause a check failure. The check for v3 also fails:

mysql> **INSERT** **INTO** **v2** **VALUES** **(2);**

ERROR 1369 (HY000): CHECK OPTION failed 'test .v2'

mysql> **INSERT** **INTO** **v3** **VALUES** **(2);**

ERROR 1369 (HY000): CHECK OPTION failed 'test.v3'

**25.5.5** **View** **Metadata**

To obtain metadata about views:

• Query the [VIEWS](#_bookmark54) table of the INFORMATION\_SCHEMA database. See [Section 26.3.48, “The](#_bookmark54) [INFORMATION\_SCHEMA VIEWS Table”](#_bookmark54) .

• Use the SHOW CREATE VIEW statement. See Section 13.7.7.13, “SHOW CREATE VIEW Statement” .

**25.6** **Stored** **Object** **Access** **Control**

Stored programs (procedures, functions, triggers, and events) and views are defined prior to use and, when referenced, execute within a security context that determines their privileges. The privileges applicable to execution of a stored object are controlled by its DEFINER attribute and SQL SECURITY characteristic.

• [The DEFINER Attribute](#_bookmark55)

• [The SQL SECURITY Characteristic](#_bookmark56)

• [Examples](#_bookmark57)

• [Orphan Stored Objects](#_bookmark58)

• [Risk-Minimization Guidelines](#_bookmark59)

**The** **DEFINER** **Attribute**

A stored object definition can include a DEFINER attribute that names a MySQL account. If a definition omits the DEFINER attribute, the default object definer is the user who creates it.

The following rules determine which accounts you can specify as the DEFINER attribute for a stored object:

• If you have the SET\_USER\_ID privilege (or the deprecated SUPER privilege), you can specify any account as the DEFINER attribute. If the account does not exist, a warning is generated. Additionally, to set a stored object DEFINER attribute to an account that has the SYSTEM\_USER privilege, you must have the SYSTEM\_USER privilege.

• Otherwise, the only permitted account is your own, specified either literally or as CURRENT\_USER or CURRENT\_USER(). You cannot set the definer to any other account.

Creating a stored object with a nonexistent DEFINER account creates an orphan object, which may have negative consequences; see [Orphan Stored Objects](#_bookmark58).

**The** **SQL** **SECURITY** **Characteristic**

For stored routines (procedures and functions) and views, the object definition can include an SQL SECURITY characteristic with a value of DEFINER or INVOKER to specify whether the object executes in definer or invoker context. If the definition omits the SQL SECURITY characteristic, the default is definer context.

Triggers and events have no SQL SECURITY characteristic and always execute in definer context. The

server invokes these objects automatically as necessary, so there is no invoking user. Definer and invoker security contexts differ as follows:

• A stored object that executes in definer security context executes with the privileges of the account named by its DEFINER attribute. These privileges may be entirely different from those of the invoking user. The invoker must have appropriate privileges to reference the object (for example, EXECUTE to call a stored procedure or SELECT to select from a view), but during object execution, the invoker's privileges are ignored and only the DEFINER account privileges matter. If the DEFINER account has few privileges, the object is correspondingly limited in the operations it can perform. If the DEFINER account is highly privileged (such as an administrative account), the object can perform powerful operations *no* *matter* *who* *invokes* *it.*

• A stored routine or view that executes in invoker security context can perform only operations for which the invoker has privileges. The DEFINER attribute has no effect on object execution.

**Examples**

Consider the following stored procedure, which is declared with SQL SECURITY DEFINER to execute in definer security context:

CREATE DEFINER = 'admin'@'localhost' PROCEDURE p1()

SQL SECURITY DEFINER

BEGIN

UPDATE t1 SET counter = counter + 1;

END;

Any user who has the EXECUTE privilege for p1 can invoke it with a CALL statement. However, when p1 executes, it does so in definer security context and thus executes with the privileges of 'admin'@'localhost', the account named as its DEFINER attribute. This account must have the EXECUTE privilege for p1 as well as the UPDATE privilege for the table t1 referenced within the object body. Otherwise, the procedure fails.

Now consider this stored procedure, which is identical to p1 except that its SQL SECURITY characteristic is INVOKER:

CREATE DEFINER = 'admin'@'localhost' PROCEDURE p2()

SQL SECURITY INVOKER

BEGIN

UPDATE t1 SET counter = counter + 1;

END;

Unlike p1, p2 executes in invoker security context and thus with the privileges of the invoking user

regardless of the DEFINER attribute value. p2 fails if the invoker lacks the EXECUTE privilege for p2 or the UPDATE privilege for the table t1.

**Orphan** **Stored** **Objects**

An orphan stored object is one for which its DEFINER attribute names a nonexistent account:

• An orphan stored object can be created by specifying a nonexistent DEFINER account at object- creation time.

• An existing stored object can become orphaned through execution of a DROP USER statement that drops the object DEFINER account, or a RENAME USER statement that renames the object DEFINER account.

An orphan stored object may be problematic in these ways:

• Because the DEFINER account does not exist, the object may not work as expected if it executes in definer security context:

• For a stored routine, an error occurs at routine execution time if the SQL SECURITY value is DEFINER but the definer account does not exist.

• For a trigger, it is not a good idea for trigger activation to occur until the account actually does exist. Otherwise, the behavior with respect to privilege checking is undefined.

• For an event, an error occurs at event execution time if the account does not exist.

• For a view, an error occurs when the view is referenced if the SQL SECURITY value is DEFINER but the definer account does not exist.

• The object may present a security risk if the nonexistent DEFINER account is subsequently re- created for a purpose unrelated to the object. In this case, the account “adopts” the object and, with

the appropriate privileges, is able to execute it even if that is not intended.

As of MySQL 8.0.22, the server imposes additional account-management security checks designed to prevent operations that (perhaps inadvertently) cause stored objects to become orphaned or that cause adoption of stored objects that are currently orphaned:

• DROP USER fails with an error if any account to be dropped is named as the DEFINER attribute for any stored object. (That is, the statement fails if dropping an account would cause a stored object to become orphaned.)

• RENAME USER fails with an error if any account to be renamed is named as the DEFINER attribute for any stored object. (That is, the statement fails if renaming an account would cause a stored object to become orphaned.)

• CREATE USER fails with an error if any account to be created is named as the DEFINER attribute for any stored object. (That is, the statement fails if creating an account would cause the account to adopt a currently orphaned stored object.)

In certain situations, it may be necessary to deliberately execute those account-management statements even when they would otherwise fail. To make this possible, if a user has the SET\_USER\_ID privilege, that privilege overrides the orphan object security checks and the statements succeed with a warning rather than failing with an error.

To obtain information about the accounts used as stored object definers in a MySQL installation, query the INFORMATION\_SCHEMA.

This query identifies which INFORMATION\_SCHEMA tables describe objects that have a DEFINER attribute:

mysql> **SELECT** **TABLE\_SCHEMA,** **TABLE\_NAME** **FROM** **INFORMATION\_SCHEMA.COLUMNS**

**WHERE** **COLUMN\_NAME** **=** **'DEFINER';**

+--------------------+------------+

| TABLE\_SCHEMA | TABLE\_NAME |

+--------------------+------------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  | | information\_schema  information\_schema  information\_schema  information\_schema | |  |  |  | | EVENTS  ROUTINES  TRIGGERS  VIEWS | |  |  |  | |

+--------------------+------------+

The result tells you which tables to query to discover which stored object DEFINER values exist and which objects have a particular DEFINER value:

• To identify which DEFINER values exist in each table, use these queries:

SELECT DISTINCT DEFINER FROM INFORMATION\_SCHEMA.EVENTS;

SELECT DISTINCT DEFINER FROM INFORMATION\_SCHEMA.ROUTINES;

SELECT DISTINCT DEFINER FROM INFORMATION\_SCHEMA.TRIGGERS;

SELECT DISTINCT DEFINER FROM INFORMATION\_SCHEMA.VIEWS;

The query results are significant for any account displayed as follows:

• If the account exists, dropping or renaming it causes stored objects to become orphaned. If you plan to drop or rename the account, consider first dropping its associated stored objects or redefining them to have a different definer.

• If the account does not exist, creating it causes it to adopt currently orphaned stored objects. If you plan to create the account, consider whether the orphaned objects should be associated with it. If not, redefine them to have a different definer.

To redefine an object with a different definer, you can use ALTER EVENT or ALTER VIEW to directly modify the DEFINER account of events and views. For stored procedures and functions and for triggers, you must drop the object and re-create it to assign a different DEFINER account

• To identify which objects have a given DEFINER account, use these queries, substituting the account of interest for *user\_name*@*host\_name*:

SELECT EVENT\_SCHEMA, EVENT\_NAME FROM INFORMATION\_SCHEMA.EVENTS

WHERE DEFINER = '*user\_name*@*host\_name*';

SELECT ROUTINE\_SCHEMA, ROUTINE\_NAME, ROUTINE\_TYPE

FROM INFORMATION\_SCHEMA.ROUTINES

WHERE DEFINER = '*user\_name*@*host\_name* ';

SELECT TRIGGER\_SCHEMA, TRIGGER\_NAME FROM INFORMATION\_SCHEMA.TRIGGERS

WHERE DEFINER = '*user\_name*@*host\_name*';

SELECT TABLE\_SCHEMA, TABLE\_NAME FROM INFORMATION\_SCHEMA.VIEWS

WHERE DEFINER = '*user\_name*@*host\_name*';

For the [ROUTINES](#_bookmark50) table, the query includes the ROUTINE\_TYPE column so that output rows distinguish whether the DEFINER is for a stored procedure or stored function.

If the account you are searching for does not exist, any objects displayed by those queries are orphan objects.

**Risk-Minimization** **Guidelines**

To minimize the risk potential for stored object creation and use, follow these guidelines:

• Do not create orphan stored objects; that is, objects for which the DEFINER attribute names a nonexistent account. Do not cause stored objects to become orphaned by dropping or renaming an account named by the DEFINER attribute of any existing object.

• For a stored routine or view, use SQL SECURITY INVOKER in the object definition when possible so that it can be used only by users with permissions appropriate for the operations performed by the object.

• If you create definer-context stored objects while using an account that has the SET\_USER\_ID privilege (or the deprecated SUPER privilege), specify an explicit DEFINER attribute that names an account possessing only the privileges required for the operations performed by the object. Specify a highly privileged DEFINER account only when absolutely necessary.

• Administrators can prevent users from creating stored objects that specify highly privileged DEFINER accounts by not granting them the SET\_USER\_ID privilege (or the deprecated SUPER privilege).

• Definer-context objects should be written keeping in mind that they may be able to access data for which the invoking user has no privileges. In some cases, you can prevent references to these objects by not granting unauthorized users particular privileges:

• A stored routine cannot be referenced by a user who does not have the EXECUTE privilege for it.

• A view cannot be referenced by a user who does not have the appropriate privilege for it (SELECT to select from it, INSERT to insert into it, and so forth).

However, no such control exists for triggers and events because they always execute in definer context. The server invokes these objects automatically as necessary, and users do not reference them directly:

• A trigger is activated by access to the table with which it is associated, even ordinary table accesses by users with no special privileges.

• An event is executed by the server on a scheduled basis.

In both cases, if the DEFINER account is highly privileged, the object may be able to perform sensitive or dangerous operations. This remains true if the privileges needed to create the object are revoked from the account of the user who created it. Administrators should be especially careful about granting users object-creation privileges.

• 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.

**25.7** **Stored** **Program** **Binary** **Logging**

The binary log contains information about SQL statements that modify database contents. This information is stored in the form of “events” that describe the modifications. (Binary log events differ from scheduled event stored objects.) The binary log has two important purposes:

• For replication, the binary log is used on source replication servers as a record of the statements to be sent to replica servers. The source sends the events contained in its binary log to its replicas, which execute those events to make the same data changes that were made on the source. See Section 17.2, “Replication Implementation” .

• Certain data recovery operations require use of the binary log. After a backup file has been restored, the events in the binary log that were recorded after the backup was made are re-executed. These events bring databases up to date from the point of the backup. See Section 7.3.2, “Using Backups for Recovery” .

However, if logging occurs at the statement level, there are certain binary logging issues with respect to stored programs (stored procedures and functions, triggers, and events):

• In some cases, a statement might affect different sets of rows on source and replica.

• Replicated statements executed on a replica are processed by the replica's applier thread. Unless you implement replication privilege checks, which are available from MySQL 8.0.18 (see

Section 17.3.3, “Replication Privilege Checks”), the applier thread has full privileges. In this situation, it is possible for a procedure to follow different execution paths on source and replica servers, so a user could write a routine containing a dangerous statement that executes only on the replica.

• If a stored program that modifies data is nondeterministic, it is not repeatable. This can result in different data on source and replica, or cause restored data to differ from the original data.

This section describes how MySQL handles binary logging for stored programs. It states the current conditions that the implementation places on the use of stored programs, and what you can do to avoid logging problems. It also provides additional information about the reasons for these conditions.

Unless noted otherwise, the remarks here assume that binary logging is enabled on the server (see Section 5.4.4, “The Binary Log” .) If the binary log is not enabled, replication is not possible, nor is the binary log available for data recovery. From MySQL 8.0, binary logging is enabled by default, and is only disabled if you specify the --skip-log-bin or --disable-log-bin option at startup.

In general, the issues described here result when binary logging occurs at the SQL statement level (statement-based binary logging). If you use row-based binary logging, the log contains changes made to individual rows as a result of executing SQL statements. When routines or triggers execute, row changes are logged, not the statements that make the changes. For stored procedures, this means

that the CALL statement is not logged. For stored functions, row changes made within the function are logged, not the function invocation. For triggers, row changes made by the trigger are logged. On the replica side, only the row changes are seen, not the stored program invocation.

Mixed format binary logging (binlog\_format=MIXED) uses statement-based binary logging, except for cases where only row-based binary logging is guaranteed to lead to proper results. With mixed format, when a stored function, stored procedure, trigger, event, or prepared statement contains anything that is not safe for statement-based binary logging, the entire statement is marked as unsafe and logged in row format. The statements used to create and drop procedures, functions, triggers, and events are always safe, and are logged in statement format. For more information about row- based, mixed, and statement-based logging, and how safe and unsafe statements are determined, see Section 17.2.1, “Replication Formats” .

The conditions on the use of stored functions in MySQL can be summarized as follows. These conditions do not apply to stored procedures or Event Scheduler events and they do not apply unless binary logging is enabled.

• To create or alter a stored function, you must have the SET\_USER\_ID privilege (or the deprecated SUPER privilege), in addition to the CREATE ROUTINE or ALTER ROUTINE privilege that is normally required. (Depending on the DEFINER value in the function definition, SET\_USER\_ID or SUPER might be required regardless of whether binary logging is enabled. See Section 13.1.17, “CREATE PROCEDURE and CREATE FUNCTION Statements” .)

• When you create a stored function, you must declare either that it is deterministic or that it does not modify data. Otherwise, it may be unsafe for data recovery or replication.

By default, for a CREATE FUNCTION statement to be accepted, at least one of DETERMINISTIC, NO SQL, or READS SQL DATA must be specified explicitly. Otherwise an error occurs:

ERROR 1418 (HY000): This function has none of DETERMINISTIC, NO SQL,

or READS SQL DATA in its declaration and binary logging is enabled

(you \*might\* want to use the less safe log\_bin\_trust\_function\_creators

variable)

This function is deterministic (and does not modify data), so it is safe:

CREATE FUNCTION f1(i INT)

RETURNS INT

DETERMINISTIC

READS SQL DATA

BEGIN

RETURN i;

END;

This function uses UUID(), which is not deterministic, so the function also is not deterministic and is not safe:

CREATE FUNCTION f2()

RETURNS CHAR(36) CHARACTER SET utf8mb4

BEGIN

RETURN UUID();

END;

This function modifies data, so it may not be safe:

CREATE FUNCTION f3(p\_id INT)

RETURNS INT

BEGIN

UPDATE t SET modtime = NOW() WHERE id = p\_id;

RETURN ROW\_COUNT();

END;

Assessment of the nature of a function is based on the “honesty” of the creator. MySQL does not check that a function declared DETERMINISTIC is free of statements that produce nondeterministic results.

• When you attempt to execute a stored function, if binlog\_format=STATEMENT is set, the DETERMINISTIC keyword must be specified in the function definition. If this is not the case, an error is generated and the function does not run, unless log\_bin\_trust\_function\_creators=1 is specified to override this check (see below). For recursive function calls, the DETERMINISTIC keyword is required on the outermost call only. If row-based or mixed binary logging is in use, the statement is accepted and replicated even if the function was defined without the DETERMINISTIC keyword.

• Because MySQL does not check if a function really is deterministic at creation time, the invocation of a stored function with the DETERMINISTIC keyword might carry out an action that is unsafe for statement-based logging, or invoke a function or procedure containing unsafe statements. If this occurs when binlog\_format=STATEMENT is set, a warning message is issued. If row-based or mixed binary logging is in use, no warning is issued, and the statement is replicated in row-based format.

• To relax the preceding conditions on function creation (that you must have the SUPER privilege and that a function must be declared deterministic or to not modify data), set the global log\_bin\_trust\_function\_creators system variable to 1. By default, this variable has a value of 0, but you can change it like this:

mysql> **SET** **GLOBAL** **log\_bin\_trust\_function\_creators** **=** **1;**

You can also set this variable at server startup.

If binary logging is not enabled, log\_bin\_trust\_function\_creators does not apply. SUPER is not required for function creation unless, as described previously, the DEFINER value in the function definition requires it.

• For information about built-in functions that may be unsafe for replication (and thus cause stored functions that use them to be unsafe as well), see Section 17.5.1, “Replication Features and Issues” .

Triggers are similar to stored functions, so the preceding remarks regarding functions also apply to triggers with the following exception: CREATE TRIGGER does not have an optional DETERMINISTIC characteristic, so triggers are assumed to be always deterministic. However, this assumption might be invalid in some cases. For example, the UUID() function is nondeterministic (and does not replicate). Be careful about using such functions in triggers.

Triggers can update tables, so error messages similar to those for stored functions occur with CREATE TRIGGER if you do not have the required privileges. On the replica side, the replica uses the trigger DEFINER attribute to determine which user is considered to be the creator of the trigger.

The rest of this section provides additional detail about the logging implementation and its implications. You need not read it unless you are interested in the background on the rationale for the current logging-related conditions on stored routine use. This discussion applies only for statement-based logging, and not for row-based logging, with the exception of the first item: CREATE and DROP statements are logged as statements regardless of the logging mode.

• The server writes CREATE EVENT, CREATE PROCEDURE, CREATE FUNCTION, ALTER EVENT, ALTER PROCEDURE, ALTER FUNCTION, DROP EVENT, DROP PROCEDURE, and DROP FUNCTION statements to the binary log.

• A stored function invocation is logged as a SELECT statement if the function changes data and occurs within a statement that would not otherwise be logged. This prevents nonreplication of data changes that result from use of stored functions in nonlogged statements. For example, SELECT statements are not written to the binary log, but a SELECT might invoke a stored function that makes changes. To handle this, a SELECT *func\_name* () statement is written to the binary log when the given function makes a change. Suppose that the following statements are executed on the source server:

CREATE FUNCTION f1(a INT) RETURNS INT

BEGIN

IF (a < 3) THEN

INSERT INTO t2 VALUES (a);

END IF;

RETURN 0;

END;

CREATE TABLE t1 (a INT);

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

SELECT f1(a) FROM t1;

When the SELECT statement executes, the function f1() is invoked three times. Two of those invocations insert a row, and MySQL logs a SELECT statement for each of them. That is, MySQL writes the following statements to the binary log:

SELECT f1(1);

SELECT f1(2);

The server also logs a SELECT statement for a stored function invocation when the function invokes a stored procedure that causes an error. In this case, the server writes the SELECT statement to the log along with the expected error code. On the replica, if the same error occurs, that is the expected result and replication continues. Otherwise, replication stops.

• Logging stored function invocations rather than the statements executed by a function has a security implication for replication, which arises from two factors:

• It is possible for a function to follow different execution paths on source and replica servers.

• Statements executed on a replica are processed by the replica's applier thread. Unless you implement replication privilege checks, which are available from MySQL 8.0.18 (see Section 17.3.3, “Replication Privilege Checks”), the applier thread has full privileges.

The implication is that although a user must have the CREATE ROUTINE privilege to create a function, the user can write a function containing a dangerous statement that executes only on the replica where it is processed by a thread that has full privileges. For example, if the source and replica servers have server ID values of 1 and 2, respectively, a user on the source server could create and invoke an unsafe function unsafe\_func() as follows:

mysql> **delimiter** **//**

mysql> **CREATE** **FUNCTION** **unsafe\_func** **()** **RETURNS** **INT**

-> **BEGIN**

-> **IF** **@@server\_id=2** **THEN** ***dangerous\_statement*** **;** **END** **IF;**

-> **RETURN** **1;**

-> **END;**

-> **//**

mysql> **delimiter** **;**

mysql> **INSERT** **INTO** **t** **VALUES(unsafe\_func());**

The CREATE FUNCTION and INSERT statements are written to the binary log, so the replica executes them. Because the replica's applier thread has full privileges, it executes the dangerous statement. Thus, the function invocation has different effects on the source and replica and is not replication-safe.

To guard against this danger for servers that have binary logging enabled, stored function creators must have the SUPER privilege, in addition to the usual CREATE ROUTINE privilege that is required. Similarly, to use ALTER FUNCTION, you must have the SUPER privilege in addition to the ALTER ROUTINE privilege. Without the SUPER privilege, an error occurs:

ERROR 1419 (HY000): You do not have the SUPER privilege and

binary logging is enabled (you \*might\* want to use the less safe

log\_bin\_trust\_function\_creators variable)

If you do not want to require function creators to have the SUPER privilege (for example, if all users with the CREATE ROUTINE privilege on your system are experienced application developers), set the global log\_bin\_trust\_function\_creators system variable to 1. You can also set this

variable at server startup. If binary logging is not enabled, log\_bin\_trust\_function\_creators does not apply. SUPER is not required for function creation unless, as described previously, the DEFINER value in the function definition requires it.

• The use of replication privilege checks where available (from MySQL 8.0.18) is recommended whatever choice you make about privileges for function creators. Replication privilege checks can be set up to ensure that only expected and relevant operations are authorized for the replication channel. For instructions to do this, see Section 17.3.3, “Replication Privilege Checks” .

• If a function that performs updates is nondeterministic, it is not repeatable. This can have two undesirable effects:

• It causes a replica to differ from the source.

• Restored data does not match the original data.

To deal with these problems, MySQL enforces the following requirement: On a source server, creation and alteration of a function is refused unless you declare the function to be deterministic or to not modify data. Two sets of function characteristics apply here:

• The DETERMINISTIC and NOT DETERMINISTIC characteristics indicate whether a function always produces the same result for given inputs. The default is NOT DETERMINISTIC if neither characteristic is given. To declare that a function is deterministic, you must specify DETERMINISTIC explicitly.

• The CONTAINS SQL, NO SQL, READS SQL DATA, and MODIFIES SQL DATA characteristics provide information about whether the function reads or writes data. Either NO SQL or READS SQL DATA indicates that a function does not change data, but you must specify one of these explicitly because the default is CONTAINS SQL if no characteristic is given.

By default, for a CREATE FUNCTION statement to be accepted, at least one of DETERMINISTIC, NO SQL, or READS SQL DATA must be specified explicitly. Otherwise an error occurs:

ERROR 1418 (HY000): This function has none of DETERMINISTIC, NO SQL,

or READS SQL DATA in its declaration and binary logging is enabled

(you \*might\* want to use the less safe log\_bin\_trust\_function\_creators

variable)

If you set log\_bin\_trust\_function\_creators to 1, the requirement that functions be deterministic or not modify data is dropped.

• Stored procedure calls are logged at the statement level rather than at the CALL level. That is, the server does not log the CALL statement, it logs those statements within the procedure that actually execute. As a result, the same changes that occur on the source server also occur on replicas. This prevents problems that could result from a procedure having different execution paths on different machines.

In general, statements executed within a stored procedure are written to the binary log using the same rules that would apply were the statements to be executed in standalone fashion. Some special care is taken when logging procedure statements because statement execution within procedures is not quite the same as in nonprocedure context:

• A statement to be logged might contain references to local procedure variables. These variables do not exist outside of stored procedure context, so a statement that refers to such a variable cannot be logged literally. Instead, each reference to a local variable is replaced by this construct for logging purposes:

NAME\_CONST(*var\_name*, *var\_value*)

*var\_name* is the local variable name, and *var\_value* is a constant indicating the value that the variable has at the time the statement is logged. NAME\_CONST() has a value of *var\_value*, and a “name” of *var\_name*. Thus, if you invoke this function directly, you get a result like this:

mysql> **SELECT** **NAME\_CONST('myname',** **14);**

+--------+

| myname |

+--------+

| 14 |

+--------+

NAME\_CONST() enables a logged standalone statement to be executed on a replica with the same effect as the original statement that was executed on the source within a stored procedure.

The use of NAME\_CONST() can result in a problem for CREATE TABLE ... SELECT statements when the source column expressions refer to local variables. Converting these references to NAME\_CONST() expressions can result in column names that are different on the source and replica servers, or names that are too long to be legal column identifiers. A workaround is to supply aliases for columns that refer to local variables. Consider this statement when myvar has a value of 1:

CREATE TABLE t1 SELECT myvar; This is rewritten as follows:

CREATE TABLE t1 SELECT NAME\_CONST(myvar, 1);

To ensure that the source and replica tables have the same column names, write the statement like this:

CREATE TABLE t1 SELECT myvar AS myvar; The rewritten statement becomes:

CREATE TABLE t1 SELECT NAME\_CONST(myvar, 1) AS myvar;

• A statement to be logged might contain references to user-defined variables. To handle this, MySQL writes a SET statement to the binary log to make sure that the variable exists on the replica with the same value as on the source. For example, if a statement refers to a variable @my\_var, that statement is preceded in the binary log by the following statement, where *value* is the value of @my\_var on the source:

SET @my\_var = *value*;

• Procedure calls can occur within a committed or rolled-back transaction. Transactional context is accounted for so that the transactional aspects of procedure execution are replicated correctly.

That is, the server logs those statements within the procedure that actually execute and modify data, and also logs BEGIN, COMMIT, and ROLLBACK statements as necessary. For example, if a procedure updates only transactional tables and is executed within a transaction that is rolled back, those updates are not logged. If the procedure occurs within a committed transaction, BEGIN and COMMIT statements are logged with the updates. For a procedure that executes within a rolled-back transaction, its statements are logged using the same rules that would apply if the statements were executed in standalone fashion:

• Updates to transactional tables are not logged.

• Updates to nontransactional tables are logged because rollback does not cancel them.

• Updates to a mix of transactional and nontransactional tables are logged surrounded by BEGIN and ROLLBACK so that replicas make the same changes and rollbacks as on the source.

• A stored procedure call is *not* written to the binary log at the statement level if the procedure is invoked from within a stored function. In that case, the only thing logged is the statement that invokes the function (if it occurs within a statement that is logged) or a DO statement (if it occurs within a statement that is not logged). For this reason, care should be exercised in the use of stored functions

that invoke a procedure, even if the procedure is otherwise safe in itself.

**25.8** **Restrictions** **on** **Stored** **Programs**

• [SQL Statements Not Permitted in Stored Routines](#_bookmark60)

• [Restrictions for Stored Functions](#_bookmark61)

• [Restrictions for Triggers](#_bookmark62)

• [Name Conflicts within Stored Routines](#_bookmark63)

• [Replication Considerations](#_bookmark64)

• [Debugging Considerations](#_bookmark65)

• [Unsupported Syntax from the SQL:2003 Standard](#_bookmark66)

• [Stored Routine Concurrency Considerations](#_bookmark67)

• [Event Scheduler Restrictions](#_bookmark68)

• [Stored routines and triggers in NDB Cluster](#_bookmark69)

These restrictions apply to the features described in [Chapter 25, *Stored* *Objects*](#_bookmark23).

Some of the restrictions noted here apply to all stored routines; that is, both to stored procedures and stored functions. There are also some [restrictions specific to stored functions](#_bookmark61) but not to stored procedures.

The restrictions for stored functions also apply to triggers. There are also some [restrictions specific to](#_bookmark62) [triggers](#_bookmark62).

The restrictions for stored procedures also apply to the DO clause of Event Scheduler event definitions. There are also some [restrictions specific to events](#_bookmark68).

**SQL** **Statements** **Not** **Permitted** **in** **Stored** **Routines**

Stored routines cannot contain arbitrary SQL statements. The following statements are not permitted:

• The locking statements LOCK TABLES and UNLOCK TABLES.

• ALTER VIEW.

• LOAD DATA and LOAD XML.

• SQL prepared statements (PREPARE, EXECUTE, DEALLOCATE PREPARE) can be used in stored procedures, but not stored functions or triggers. Thus, stored functions and triggers cannot use dynamic SQL (where you construct statements as strings and then execute them).

• Generally, statements not permitted in SQL prepared statements are also not permitted in stored programs. For a list of statements supported as prepared statements, see Section 13.5, “Prepared Statements” . Exceptions are SIGNAL, RESIGNAL, and GET DIAGNOSTICS, which are not permissible as prepared statements but are permitted in stored programs.

• Because local variables are in scope only during stored program execution, references to them are not permitted in prepared statements created within a stored program. Prepared statement scope is the current session, not the stored program, so the statement could be executed after the program ends, at which point the variables would no longer be in scope. For example, SELECT ... INTO *local\_var* cannot be used as a prepared statement. This restriction also applies to stored procedure and function parameters. See Section 13.5.1, “PREPARE Statement” .

• Within all stored programs (stored procedures and functions, triggers, and events), the parser treats BEGIN [WORK] as the beginning of a BEGIN ... END block. To begin a transaction in this context,

use START TRANSACTION instead.

**Restrictions** **for** **Stored** **Functions**

The following additional statements or operations are not permitted within stored functions. They are permitted within stored procedures, except stored procedures that are invoked from within a stored function or trigger. For example, if you use FLUSH in a stored procedure, that stored procedure cannot be called from a stored function or trigger.

• 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. This includes SELECT statements that do not have an INTO *var\_list* clause and other statements such as SHOW, EXPLAIN, and CHECK TABLE. A function can process a result set either with SELECT ... INTO *var\_list* or by using a cursor and FETCH statements. See Section 13.2.13.1, “SELECT ... INTO Statement” , and Section 13.6.6, “Cursors” .

• FLUSH statements.

• Stored functions cannot be used recursively.

• A stored function or trigger cannot modify a table that is already being used (for reading or writing) by the statement that invoked the function or trigger.

• If you refer to a temporary table multiple times in a stored function under different aliases, a Can't reopen table: '*tbl\_name*' error occurs, even if the references occur in different statements within the function.

• HANDLER ... READ statements that invoke stored functions can cause replication errors and are disallowed.

**Restrictions** **for** **Triggers**

For triggers, the following additional restrictions apply:

• Triggers are not activated by foreign key actions.

• When using row-based replication, triggers on the replica are not activated by statements originating on the source. The triggers on the replica are activated when using statement-based replication. For more information, see Section 17.5.1.36, “Replication and Triggers” .

• The RETURN statement is not permitted in triggers, which cannot return a value. To exit a trigger immediately, use the LEAVE statement.

• Triggers are not permitted on tables in the mysql database. Nor are they permitted on INFORMATION\_SCHEMA or performance\_schema tables. Those tables are actually views and triggers are not permitted on views.

• The trigger cache does not detect when metadata of the underlying objects has changed. If a trigger uses a table and the table has changed since the trigger was loaded into the cache, the trigger operates using the outdated metadata.

**Name** **Conflicts** **within** **Stored** **Routines**

The same identifier might be used for a routine parameter, a local variable, and a table column. Also, the same local variable name can be used in nested blocks. For example:

CREATE PROCEDURE p (i INT)

BEGIN

DECLARE i INT DEFAULT 0;

SELECT i FROM t;

BEGIN

DECLARE i INT DEFAULT 1;

SELECT i FROM t;

END;

END;

In such cases, the identifier is ambiguous and the following precedence rules apply:

• A local variable takes precedence over a routine parameter or table column.

• A routine parameter takes precedence over a table column.

• A local variable in an inner block takes precedence over a local variable in an outer block. The behavior that variables take precedence over table columns is nonstandard.

**Replication** **Considerations**

Use of stored routines can cause replication problems. This issue is discussed further in [Section 25.7,](#_bookmark47) [“Stored Program Binary Logging”](#_bookmark47) .

The --replicate-wild-do-table=*db\_name.tbl\_name* option applies to tables, views, and triggers. It does not apply to stored procedures and functions, or events. To filter statements operating on the latter objects, use one or more of the --replicate-\*-db options.

**Debugging** **Considerations**

There are no stored routine debugging facilities.

**Unsupported** **Syntax** **from** **the** **SQL:2003** **Standard**

The MySQL stored routine syntax is based on the SQL:2003 standard. The following items from that standard are not currently supported:

• UNDO handlers

• FOR loops

**Stored** **Routine** **Concurrency** **Considerations**

To prevent problems of interaction between sessions, when a client issues a statement, the server uses a snapshot of routines and triggers available for execution of the statement. That is, the server calculates a list of procedures, functions, and triggers that may be used during execution of the statement, loads them, and then proceeds to execute the statement. While the statement executes, it does not see changes to routines performed by other sessions.

For maximum concurrency, stored functions should minimize their side-effects; in particular, updating a table within a stored function can reduce concurrent operations on that table. A stored function acquires table locks before executing, to avoid inconsistency in the binary log due to mismatch of the order in which statements execute and when they appear in the log. When statement-based binary logging is used, statements that invoke a function are recorded rather than the statements executed within the function. Consequently, stored functions that update the same underlying tables do not execute in parallel. In contrast, stored procedures do not acquire table-level locks. All statements executed within stored procedures are written to the binary log, even for statement-based binary logging. See [Section 25.7, “Stored Program Binary Logging”](#_bookmark47) .

**Event** **Scheduler** **Restrictions**

The following limitations are specific to the Event Scheduler:

• Event names are handled in case-insensitive fashion. For example, you cannot have two events in the same database with the names anEvent and AnEvent.



• An event may not be created from within a stored program. An event may not be altered, or dropped from within a stored program, if the event name is specified by means of a variable. An event also may not create, alter, or drop stored routines or triggers.

• DDL statements on events are prohibited while a LOCK TABLES statement is in effect.

• Event timings using the intervals YEAR, QUARTER, MONTH, and YEAR\_MONTH are resolved in months; those using any other interval are resolved in seconds. There is no way to cause events scheduled to occur at the same second to execute in a given order. In addition—due to rounding, the nature of threaded applications, and the fact that a nonzero length of time is required to create events and to signal their execution—events may be delayed by as much as 1 or 2 seconds. However, the time shown in the Information Schema [EVENTS](#_bookmark53) table's LAST\_EXECUTED column is always accurate to within one second of the actual event execution time. (See also Bug #16522.)

• Each execution of the statements contained in the body of an event takes place in a new connection; thus, these statements have no effect in a given user session on the server's statement counts such as Com\_select and Com\_insert that are displayed by SHOW STATUS. However, such counts *are* updated in the global scope. (Bug #16422)

• Events do not support times later than the end of the Unix Epoch; this is approximately the beginning of the year 2038. Such dates are specifically not permitted by the Event Scheduler. (Bug #16396)

• References to stored functions, loadable functions, and tables in the ON SCHEDULE clauses of CREATE EVENT and ALTER EVENT statements are not supported. These sorts of references are not permitted. (See Bug #22830 for more information.)

**Stored** **routines** **and** **triggers** **in** **NDB** **Cluster**

While stored procedures, stored functions, triggers, and scheduled events are all supported by tables using the NDB storage engine, you must keep in mind that these do *not* propagate automatically between MySQL Servers acting as Cluster SQL nodes. This is because stored routine and trigger definitions are stored in tables in the mysql system database using InnoDB tables, which are not copied between Cluster nodes.

Any stored routine or trigger that interacts with MySQL Cluster tables must be re-created by running the appropriate CREATE PROCEDURE, CREATE FUNCTION, or CREATE TRIGGER statements on each MySQL Server that participates in the cluster where you wish to use the stored routine or trigger. Similarly, any changes to existing stored routines or triggers must be carried out explicitly on all Cluster SQL nodes, using the appropriate ALTER or DROP statements on each MySQL Server accessing the cluster.

**Warning**

Do *not* attempt to work around the issue just described by converting any mysql database tables to use the NDB storage engine. *Altering* *the* *system* *tables* *in* *the* *mysql* *database* *is* *not* *supported* and is very likely to produce undesirable results.

**25.9** **Restrictions** **on** **Views**

The maximum number of tables that can be referenced in the definition of a view is 61.

View processing is not optimized:

• It is not possible to create an index on a view.

• Indexes can be used for views processed using the merge algorithm. However, a view that is processed with the temptable algorithm is unable to take advantage of indexes on its underlying tables (although indexes can be used during generation of the temporary tables).

There is a general principle that you cannot modify a table and select from the same table in a subquery. See Section 13.2.15.12, “Restrictions on Subqueries” .

The same principle also applies if you select from a view that selects from the table, if the view selects from the table in a subquery and the view is evaluated using the merge algorithm. Example:

CREATE VIEW v1 AS

SELECT \* FROM t2 WHERE EXISTS (SELECT 1 FROM t1 WHERE t1.a = t2.a);

UPDATE t1, v2 SET t1.a = 1 WHERE t1.b = v2.b;

If the view is evaluated using a temporary table, you *can* select from the table in the view subquery and still modify that table in the outer query. In this case, the view is stored in a temporary table and thus you are not really selecting from the table in a subquery and modifying it at the same time. (This is another reason you might wish to force MySQL to use the temptable algorithm by specifying ALGORITHM = TEMPTABLE in the view definition.)

You can use DROP TABLE or ALTER TABLE to drop or alter a table that is used in a view definition. No warning results from the DROP or ALTER operation, even though this invalidates the view. Instead, an error occurs later, when the view is used. CHECK TABLE can be used to check for views that have been invalidated by DROP or ALTER operations.

With regard to view updatability, the overall goal for views is that if any view is theoretically updatable, it should be updatable in practice. Many theoretically updatable views can be updated now, but limitations still exist. For details, see [Section 25.5.3, “Updatable and Insertable Views”](#_bookmark43) .

There exists a shortcoming with the current implementation of views. If a user is granted the basic privileges necessary to create a view (the CREATE VIEW and SELECT privileges), that user cannot call SHOW CREATE VIEW on that object unless the user is also granted the SHOW VIEW privilege.

That shortcoming can lead to problems backing up a database with mysqldump, which may fail due to insufficient privileges. This problem is described in Bug #22062.

The workaround to the problem is for the administrator to manually grant the SHOW VIEW privilege to users who are granted CREATE VIEW, since MySQL doesn't grant it implicitly when views are created.

Views do not have indexes, so index hints do not apply. Use of index hints when selecting from a view is not permitted.

SHOW CREATE VIEW displays view definitions using an AS *alias\_name* clause for each column. If a column is created from an expression, the default alias is the expression text, which can be quite long. Aliases for column names in CREATE VIEW statements are checked against the maximum column length of 64 characters (not the maximum alias length of 256 characters). As a result, views created from the output of SHOW CREATE VIEW fail if any column alias exceeds 64 characters. This can cause problems in the following circumstances for views with too-long aliases:

• View definitions fail to replicate to newer replicas that enforce the column-length restriction.

• Dump files created with mysqldump cannot be loaded into servers that enforce the column-length restriction.

A workaround for either problem is to modify each problematic view definition to use aliases that provide shorter column names. Then the view replicates properly, and can be dumped and reloaded without causing an error. To modify the definition, drop and create the view again with DROP VIEW and CREATE VIEW, or replace the definition with CREATE OR REPLACE VIEW.

For problems that occur when reloading view definitions in dump files, another workaround is to edit the dump file to modify its CREATE VIEW statements. However, this does not change the original view definitions, which may cause problems for subsequent dump operations.

Chapter 26 INFORMATION\_SCHEMA Tables

**Table** **of** **Contents**

[26.1 Introduction](#_bookmark70) [4838](#_bookmark70)

[26.2 INFORMATION\_SCHEMA Table Reference](#_bookmark71) [4841](#_bookmark71)

[26.3 INFORMATION\_SCHEMA General Tables](#_bookmark72) [4846](#_bookmark72)

[26.3.1 INFORMATION\_SCHEMA General Table Reference](#_bookmark73) [4846](#_bookmark73)

[26.3.2 The INFORMATION\_SCHEMA ADMINISTRABLE\_ROLE\_AUTHORIZATIONS Table](#_bookmark74) [4848](#_bookmark74)

[26.3.3 The INFORMATION\_SCHEMA APPLICABLE\_ROLES Table](#_bookmark75) [4848](#_bookmark75)

[26.3.4 The INFORMATION\_SCHEMA CHARACTER\_SETS Table](#_bookmark76) [4849](#_bookmark76)

[26.3.5 The INFORMATION\_SCHEMA CHECK\_CONSTRAINTS Table](#_bookmark77) [4849](#_bookmark77)

[26.3.6 The INFORMATION\_SCHEMA COLLATIONS Table](#_bookmark78) [4850](#_bookmark78)

[26.3.7 The INFORMATION\_SCHEMA COLLATION\_CHARACTER\_SET\_APPLICABILITY](#_bookmark79)

[Table](#_bookmark79) [4851](#_bookmark79)

[26.3.8 The INFORMATION\_SCHEMA COLUMNS Table](#_bookmark80) [4851](#_bookmark80)

[26.3.9 The INFORMATION\_SCHEMA COLUMNS\_EXTENSIONS Table](#_bookmark81) [4854](#_bookmark81)

[26.3.10 The INFORMATION\_SCHEMA COLUMN\_PRIVILEGES Table](#_bookmark82) [4854](#_bookmark82)

[26.3.11 The INFORMATION\_SCHEMA COLUMN\_STATISTICS Table](#_bookmark83) [4855](#_bookmark83)

[26.3.12 The INFORMATION\_SCHEMA ENABLED\_ROLES Table](#_bookmark84) [4855](#_bookmark84)

[26.3.13 The INFORMATION\_SCHEMA ENGINES Table](#_bookmark85) [4856](#_bookmark85)

[26.3.14 The INFORMATION\_SCHEMA EVENTS Table](#_bookmark53) [4857](#_bookmark53)

[26.3.15 The INFORMATION\_SCHEMA FILES Table](#_bookmark86) [4860](#_bookmark86)

[26.3.16 The INFORMATION\_SCHEMA KEY\_COLUMN\_USAGE Table](#_bookmark87) [4868](#_bookmark87)

[26.3.17 The INFORMATION\_SCHEMA KEYWORDS Table](#_bookmark88) [4869](#_bookmark88)

[26.3.18 The INFORMATION\_SCHEMA Table](#_bookmark89)ndbtransidmysqlconnectionmap . . . . . . . . . . . . [4869](#_bookmark89)

[26.3.19 The INFORMATION\_SCHEMA OPTIMIZER\_TRACE Table](#_bookmark90) [4871](#_bookmark90)

[26.3.20 The INFORMATION\_SCHEMA PARAMETERS Table](#_bookmark91) [4871](#_bookmark91)

[26.3.21 The INFORMATION\_SCHEMA PARTITIONS Table](#_bookmark3) [4872](#_bookmark3)

[26.3.22 The INFORMATION\_SCHEMA PLUGINS Table](#_bookmark92) [4875](#_bookmark92)

[26.3.23 The INFORMATION\_SCHEMA PROCESSLIST Table](#_bookmark93) [4877](#_bookmark93)

[26.3.24 The INFORMATION\_SCHEMA PROFILING Table](#_bookmark94) [4878](#_bookmark94)

[26.3.25 The INFORMATION\_SCHEMA REFERENTIAL\_CONSTRAINTS Table](#_bookmark95) [4879](#_bookmark95)

[26.3.26 The INFORMATION\_SCHEMA RESOURCE\_GROUPS Table](#_bookmark96) [4880](#_bookmark96)

[26.3.27 The INFORMATION\_SCHEMA ROLE\_COLUMN\_GRANTS Table](#_bookmark97) [4880](#_bookmark97)

[26.3.28 The INFORMATION\_SCHEMA ROLE\_ROUTINE\_GRANTS Table](#_bookmark98) [4881](#_bookmark98)

[26.3.29 The INFORMATION\_SCHEMA ROLE\_TABLE\_GRANTS Table](#_bookmark99) [4882](#_bookmark99)

[26.3.30 The INFORMATION\_SCHEMA ROUTINES Table](#_bookmark50) [4883](#_bookmark50)

[26.3.31 The INFORMATION\_SCHEMA SCHEMATA Table](#_bookmark100) [4885](#_bookmark100)

[26.3.32 The INFORMATION\_SCHEMA SCHEMATA\_EXTENSIONS Table](#_bookmark101) [4886](#_bookmark101)

[26.3.33 The INFORMATION\_SCHEMA SCHEMA\_PRIVILEGES Table](#_bookmark102) [4887](#_bookmark102)

[26.3.34 The INFORMATION\_SCHEMA STATISTICS Table](#_bookmark103) [4887](#_bookmark103)

[26.3.35 The INFORMATION\_SCHEMA ST\_GEOMETRY\_COLUMNS Table](#_bookmark104) [4890](#_bookmark104)

[26.3.36 The INFORMATION\_SCHEMA ST\_SPATIAL\_REFERENCE\_SYSTEMS Table](#_bookmark105) [4890](#_bookmark105)

[26.3.37 The INFORMATION\_SCHEMA ST\_UNITS\_OF\_MEASURE Table](#_bookmark106) [4892](#_bookmark106)

[26.3.38 The INFORMATION\_SCHEMA TABLES Table](#_bookmark107) [4892](#_bookmark107)

[26.3.39 The INFORMATION\_SCHEMA TABLES\_EXTENSIONS Table](#_bookmark108) [4896](#_bookmark108)

[26.3.40 The INFORMATION\_SCHEMA TABLESPACES Table](#_bookmark109) [4896](#_bookmark109)

[26.3.41 The INFORMATION\_SCHEMA TABLESPACES\_EXTENSIONS Table](#_bookmark110) [4897](#_bookmark110)

[26.3.42 The INFORMATION\_SCHEMA TABLE\_CONSTRAINTS Table](#_bookmark111) [4897](#_bookmark111)

[26.3.43 The INFORMATION\_SCHEMA TABLE\_CONSTRAINTS\_EXTENSIONS Table](#_bookmark112) [4898](#_bookmark112)

[26.3.44 The INFORMATION\_SCHEMA TABLE\_PRIVILEGES Table](#_bookmark113) [4898](#_bookmark113)

[26.3.45 The INFORMATION\_SCHEMA TRIGGERS Table](#_bookmark52) [4899](#_bookmark52)

[26.3.46 The INFORMATION\_SCHEMA USER\_ATTRIBUTES Table](#_bookmark114) [4901](#_bookmark114)

[26.3.47 The INFORMATION\_SCHEMA USER\_PRIVILEGES Table](#_bookmark115) [4902](#_bookmark115)

[26.3.48 The INFORMATION\_SCHEMA VIEWS Table](#_bookmark54) [4902](#_bookmark54)

[26.3.49 The INFORMATION\_SCHEMA VIEW\_ROUTINE\_USAGE Table](#_bookmark116) [4904](#_bookmark116)

[26.3.50 The INFORMATION\_SCHEMA VIEW\_TABLE\_USAGE Table](#_bookmark117) [4904](#_bookmark117)

[26.4 INFORMATION\_SCHEMA InnoDB Tables](#_bookmark118) [4905](#_bookmark118)

[26.4.1 INFORMATION\_SCHEMA InnoDB Table Reference](#_bookmark119) [4905](#_bookmark119)

[26.4.2 The INFORMATION\_SCHEMA INNODB\_BUFFER\_PAGE Table](#_bookmark120) [4906](#_bookmark120)

[26.4.3 The INFORMATION\_SCHEMA INNODB\_BUFFER\_PAGE\_LRU Table](#_bookmark121) [4910](#_bookmark121)

[26.4.4 The INFORMATION\_SCHEMA INNODB\_BUFFER\_POOL\_STATS Table](#_bookmark122) [4913](#_bookmark122)

[26.4.5 The INFORMATION\_SCHEMA INNODB\_CACHED\_INDEXES Table](#_bookmark123) [4916](#_bookmark123)

[26.4.6 The INFORMATION\_SCHEMA INNODB\_CMP INNODB\_CMP\_RESET Tables](#_bookmark124)and [4917](#_bookmark124)

[26.4.7 The INFORMATION\_SCHEMA INNODB\_CMPMEM](#_bookmark125)and

[INNODB\_CMPMEM\_RESET Tables](#_bookmark125) [4918](#_bookmark125)

[26.4.8 The INFORMATION\_SCHEMA INNODB\_CMP\_PER\_INDEX](#_bookmark126)and

[INNODB\_CMP\_PER\_INDEX\_RESET Tables](#_bookmark126) [4920](#_bookmark126)

[26.4.9 The INFORMATION\_SCHEMA INNODB\_COLUMNS Table](#_bookmark127) [4921](#_bookmark127)

[26.4.10 The INFORMATION\_SCHEMA INNODB\_DATAFILES Table](#_bookmark128) [4923](#_bookmark128)

[26.4.11 The INFORMATION\_SCHEMA INNODB\_FIELDS Table](#_bookmark129) [4923](#_bookmark129)

[26.4.12 The INFORMATION\_SCHEMA INNODB\_FOREIGN Table](#_bookmark130) [4924](#_bookmark130)

[26.4.13 The INFORMATION\_SCHEMA INNODB\_FOREIGN\_COLS Table](#_bookmark131) [4925](#_bookmark131)

[26.4.14 The INFORMATION\_SCHEMA INNODB\_FT\_BEING\_DELETED Table](#_bookmark132) [4925](#_bookmark132)

[26.4.15 The INFORMATION\_SCHEMA INNODB\_FT\_CONFIG Table](#_bookmark133) [4926](#_bookmark133)

[26.4.16 The INFORMATION\_SCHEMA INNODB\_FT\_DEFAULT\_STOPWORD Table](#_bookmark134) [4927](#_bookmark134)

[26.4.17 The INFORMATION\_SCHEMA INNODB\_FT\_DELETED Table](#_bookmark135) [4928](#_bookmark135)

[26.4.18 The INFORMATION\_SCHEMA INNODB\_FT\_INDEX\_CACHE Table](#_bookmark136) [4929](#_bookmark136)

[26.4.19 The INFORMATION\_SCHEMA INNODB\_FT\_INDEX\_TABLE Table](#_bookmark137) [4930](#_bookmark137)

[26.4.20 The INFORMATION\_SCHEMA INNODB\_INDEXES Table](#_bookmark138) [4932](#_bookmark138)

[26.4.21 The INFORMATION\_SCHEMA INNODB\_METRICS Table](#_bookmark139) [4933](#_bookmark139)

[26.4.22 The INFORMATION\_SCHEMA INNODB\_SESSION\_TEMP\_TABLESPACES Table](#_bookmark140) [4935](#_bookmark140)

[26.4.23 The INFORMATION\_SCHEMA INNODB\_TABLES Table](#_bookmark141) [4936](#_bookmark141)

[26.4.24 The INFORMATION\_SCHEMA INNODB\_TABLESPACES Table](#_bookmark142) [4937](#_bookmark142)

[26.4.25 The INFORMATION\_SCHEMA INNODB\_TABLESPACES\_BRIEF Table](#_bookmark143) [4940](#_bookmark143)

[26.4.26 The INFORMATION\_SCHEMA INNODB\_TABLESTATS View](#_bookmark144) [4940](#_bookmark144)

[26.4.27 The INFORMATION\_SCHEMA INNODB\_TEMP\_TABLE\_INFO Table](#_bookmark145) [4942](#_bookmark145)

[26.4.28 The INFORMATION\_SCHEMA INNODB\_TRX Table](#_bookmark146) [4943](#_bookmark146)

[26.4.29 The INFORMATION\_SCHEMA INNODB\_VIRTUAL Table](#_bookmark147) [4945](#_bookmark147)

[26.5 INFORMATION\_SCHEMA Thread Pool Tables](#_bookmark148) [4947](#_bookmark148)

[26.5.1 INFORMATION\_SCHEMA Thread Pool Table Reference](#_bookmark149) [4947](#_bookmark149)

[26.5.2 The INFORMATION\_SCHEMA TP\_THREAD\_GROUP\_STATE Table](#_bookmark150) [4947](#_bookmark150)

[26.5.3 The INFORMATION\_SCHEMA TP\_THREAD\_GROUP\_STATS Table](#_bookmark151) [4948](#_bookmark151)

[26.5.4 The INFORMATION\_SCHEMA TP\_THREAD\_STATE Table](#_bookmark152) [4948](#_bookmark152)

[26.6 INFORMATION\_SCHEMA Connection-Control Tables](#_bookmark153) [4949](#_bookmark153)

[26.6.1 INFORMATION\_SCHEMA Connection-Control Table Reference](#_bookmark154) [4949](#_bookmark154)

[26.6.2 The INFORMATION\_SCHEMA](#_bookmark155)

[CONNECTION\_CONTROL\_FAILED\_LOGIN\_ATTEMPTS Table](#_bookmark155) [4949](#_bookmark155)

[26.7 INFORMATION\_SCHEMA MySQL Enterprise Firewall Tables](#_bookmark156) [4949](#_bookmark156)

[26.7.1 INFORMATION\_SCHEMA Firewall Table Reference](#_bookmark157) [4949](#_bookmark157)

[26.7.2 The INFORMATION\_SCHEMA MYSQL\_FIREWALL\_USERS Table](#_bookmark158) [4950](#_bookmark158)

[26.7.3 The INFORMATION\_SCHEMA MYSQL\_FIREWALL\_WHITELIST Table](#_bookmark159) [4950](#_bookmark159)

[26.8 Extensions SHOW Statements](#_bookmark160)to [4950](#_bookmark160)

INFORMATION\_SCHEMA provides access to database *metadata*, information about the MySQL server such as the name of a database or table, the data type of a column, or access privileges. Other terms that are sometimes used for this information are *data* *dictionary* and *system* *catalog*.

**26.1** **Introduction**

INFORMATION\_SCHEMA provides access to database *metadata*, information about the MySQL server such as the name of a database or table, the data type of a column, or access privileges. Other terms that are sometimes used for this information are *data* *dictionary* and *system* *catalog*.

• [INFORMATION\_SCHEMA Usage Notes](#_bookmark161)

• [Character Set Considerations](#_bookmark162)

• [INFORMATION\_SCHEMA as Alternative to SHOW Statements](#_bookmark163)

• [INFORMATION\_SCHEMA and Privileges](#_bookmark164)

• [Performance Considerations](#_bookmark165)

• [Standards Considerations](#_bookmark166)

• [Conventions in the INFORMATION\_SCHEMA Reference Sections](#_bookmark167)

• [Related Information](#_bookmark168)

**INFORMATION\_SCHEMA** **Usage** **Notes**

INFORMATION\_SCHEMA is a database within each MySQL instance, the place that stores information about all the other databases that the MySQL server maintains. The INFORMATION\_SCHEMA database contains several read-only tables. They are actually views, not base tables, so there are no files associated with them, and you cannot set triggers on them. Also, there is no database directory with that name.

Although you can select INFORMATION\_SCHEMA as the default database with a USE statement, you

can only read the contents of tables, not perform INSERT, UPDATE, or DELETE operations on them. Here is an example of a statement that retrieves information from INFORMATION\_SCHEMA:

mysql> **SELECT** **table\_name,** **table\_type,** **engine**

**FROM** **information\_schema.tables**

**WHERE** **table\_schema** **=** **'db5'**

**ORDER** **BY** **table\_name;**

+------------+------------+--------+

| table\_name | table\_type | engine |

+------------+------------+--------+

| fk | BASE TABLE | InnoDB |

| fk2 | BASE TABLE | InnoDB |

| goto | BASE TABLE | MyISAM |

| into | BASE TABLE | MyISAM |

| k | BASE TABLE | MyISAM |

| kurs | BASE TABLE | MyISAM |

| loop | BASE TABLE | MyISAM |

| pk | BASE TABLE | InnoDB |

| t | BASE TABLE | MyISAM |

| t2 | BASE TABLE | MyISAM |

| t3 | BASE TABLE | MyISAM |

| t7 | BASE TABLE | MyISAM |

| tables | BASE TABLE | MyISAM |

| v | VIEW | NULL |

| v2 | VIEW | NULL |

| v3 | VIEW | NULL |

| v56 | VIEW | NULL |

+------------+------------+--------+

17 rows in set (0.01 sec)

Explanation: The statement requests a list of all the tables in database db5, showing just three pieces

of information: the name of the table, its type, and its storage engine. Beginning with MySQL 8.0.30, information about generated invisible primary keys is visible by default in all INFORMATION\_SCHEMA tables describing table columns, keys, or both, such as the [COLUMNS](#_bookmark80) and [STATISTICS](#_bookmark103) tables. If you wish to make such information hidden from queries that select from these tables, you can do so by setting the value of the show\_gipk\_in\_create\_table\_and\_information\_schema server system variable to OFF. For more information, see Section 13. 1.20. 11, “Generated Invisible Primary Keys” .

**Character** **Set** **Considerations**

The definition for character columns (for example, TABLES.TABLE\_NAME) is generally VARCHAR(*N*) CHARACTER SET utf8mb3 where *N* is at least 64. MySQL uses the default collation for this character set (utf8mb3\_general\_ci) for all searches, sorts, comparisons, and other string operations on such columns.

Because some MySQL objects are represented as files, searches in INFORMATION\_SCHEMA string columns can be affected by file system case sensitivity. For more information, see Section 10.8.7, “Using Collation in INFORMATION\_SCHEMA Searches” .

**INFORMATION\_SCHEMA** **as** **Alternative** **to** **SHOW** **Statements**

The SELECT ... FROM INFORMATION\_SCHEMA statement is intended as a more consistent way to provide access to the information provided by the various SHOW statements that MySQL supports (SHOW DATABASES, SHOW TABLES, and so forth). Using SELECT has these advantages, compared to SHOW:

• It conforms to Codd's rules, because all access is done on tables.

• You can use the familiar syntax of the SELECT statement, and only need to learn some table and column names.

• The implementor need not worry about adding keywords.

• You can filter, sort, concatenate, and transform the results from INFORMATION\_SCHEMA queries into whatever format your application needs, such as a data structure or a text representation to parse.

• This technique is more interoperable with other database systems. For example, Oracle Database users are familiar with querying tables in the Oracle data dictionary.

Because SHOW is familiar and widely used, the SHOW statements remain as an alternative. In fact, along with the implementation of INFORMATION\_SCHEMA, there are enhancements to SHOW as described in [Section 26.8, “Extensions to SHOW Statements”](#_bookmark160) .

**INFORMATION\_SCHEMA** **and** **Privileges**

For most INFORMATION\_SCHEMA tables, each MySQL user has the right to access them, but can see only the rows in the tables that correspond to objects for which the user has the proper access privileges. In some cases (for example, the ROUTINE\_DEFINITION column in the INFORMATION\_SCHEMA [ROUTINES](#_bookmark50) table), users who have insufficient privileges see NULL. Some tables have different privilege requirements; for these, the requirements are mentioned in the applicable table descriptions. For example, InnoDB tables (tables with names that begin with INNODB\_) require the PROCESS privilege.

The same privileges apply to selecting information from INFORMATION\_SCHEMA and viewing the same information through SHOW statements. In either case, you must have some privilege on an object to see information about it.

**Performance** **Considerations**

INFORMATION\_SCHEMA queries that search for information from more than one database might take a long time and impact performance. To check the efficiency of a query, you can use EXPLAIN. For information about using EXPLAIN output to tune INFORMATION\_SCHEMA queries, see Section 8.2.3, “Optimizing INFORMATION\_SCHEMA Queries” .

**Standards** **Considerations**

The implementation for the INFORMATION\_SCHEMA table structures in MySQL follows the ANSI/ISO SQL:2003 standard Part 11 *Schemata*. Our intent is approximate compliance with SQL:2003 core feature F021 *Basic* *information* *schema*.

Users of SQL Server 2000 (which also follows the standard) may notice a strong similarity. However, MySQL has omitted many columns that are not relevant for our implementation, and added columns that are MySQL-specific. One such added column is the ENGINE column in the INFORMATION\_SCHEMA [TABLES](#_bookmark107) table.

Although other DBMSs use a variety of names, like syscat or system, the standard name is

INFORMATION\_SCHEMA.

To avoid using any name that is reserved in the standard or in DB2, SQL Server, or Oracle, we changed the names of some columns marked “MySQL extension” . (For example, we changed COLLATION to TABLE\_COLLATION in the [TABLES](#_bookmark107) table.) See the list of reserved words near the end of this article: [https://web.archive.org/web/20070428032454/http://www.dbazine.com/db2/db2-](https://web.archive.org/web/20070428032454/http://www.dbazine.com/db2/db2-disarticles/gulutzan5) [disarticles/gulutzan5](https://web.archive.org/web/20070428032454/http://www.dbazine.com/db2/db2-disarticles/gulutzan5).

**Conventions** **in** **the** **INFORMATION\_SCHEMA** **Reference** **Sections**

The following sections describe each of the tables and columns in INFORMATION\_SCHEMA. For each column, there are three pieces of information:

• “INFORMATION\_SCHEMA Name” indicates the name for the column in the INFORMATION\_SCHEMA table. This corresponds to the standard SQL name unless the “Remarks” field says “MySQL extension.”

• “SHOW Name” indicates the equivalent field name in the closest SHOW statement, if there is one.

• “Remarks” provides additional information where applicable. If this field is NULL, it means that the value of the column is always NULL. If this field says “MySQL extension,” the column is a MySQL extension to standard SQL.

Many sections indicate what SHOW statement is equivalent to a SELECT that retrieves information from INFORMATION\_SCHEMA. For SHOW statements that display information for the default database if you omit a FROM *db\_name* clause, you can often select information for the default database by adding an AND TABLE\_SCHEMA = SCHEMA() condition to the WHERE clause of a query that retrieves information from an INFORMATION\_SCHEMA table.

**Related** **Information**

These sections discuss additional INFORMATION\_SCHEMA-related topics:

• information about INFORMATION\_SCHEMA tables specific to the InnoDB storage engine:

[Section 26.4, “INFORMATION\_SCHEMA InnoDB Tables”](#_bookmark118)

• information about INFORMATION\_SCHEMA tables specific to the thread pool plugin: [Section 26.5,](#_bookmark148) [“INFORMATION\_SCHEMA Thread Pool Tables”](#_bookmark148)

• information about INFORMATION\_SCHEMA tables specific to the CONNECTION\_CONTROL plugin:

[Section 26.6, “INFORMATION\_SCHEMA Connection-Control Tables”](#_bookmark153)

• Answers to questions that are often asked concerning the INFORMATION\_SCHEMA database:

Section A.7, “MySQL 8.0 FAQ: INFORMATION\_SCHEMA”

• INFORMATION\_SCHEMA queries and the optimizer: Section 8.2.3, “Optimizing INFORMATION\_SCHEMA Queries”

• The effect of collation on INFORMATION\_SCHEMA comparisons: Section 10.8.7, “Using Collation in INFORMATION\_SCHEMA Searches”

**26.2** **INFORMATION\_SCHEMA** **Table** **Reference**

The following table summarizes all available INFORMATION\_SCHEMA tables. For greater detail, see the individual table descriptions.

**Table** **26.1** **INFORMATION\_SCHEMA** **Tables**

|  |  |  |  |
| --- | --- | --- | --- |
| **Table** **Name** | **Description** | **Introduced** | **Deprecated** |
| ADMINISTRABLE\_ROLE\_ | tlIeZsTeIrNo roles for current user or role | 8.0.19 |  |
| [APPLICABLE\_ROLES](#_bookmark75) | Applicable roles for  current user | 8.0.19 |  |
| [CHARACTER\_SETS](#_bookmark76) | Available character sets |  |  |
| [CHECK\_CONSTRAINTS](#_bookmark77) | Table and column  CHECK constraints | 8.0.16 |  |
| COLLATION\_CHARACTER | C\_ShacAt\_rPeItCaApBpIliaIl to each collation |  |  |
| [COLLATIONS](#_bookmark78) | Collations for each  character set |  |  |
| [COLUMN\_PRIVILEGES](#_bookmark82) | Privileges defined on columns |  |  |
| [COLUMN\_STATISTICS](#_bookmark83) | Histogram statistics for column values |  |  |
| [COLUMNS](#_bookmark80) | Columns in each table |  |  |
| [COLUMNS\_EXTENSIONS](#_bookmark81) | Column attributes for primary and secondary storage engines | 8.0.21 |  |
| CONNECTION\_CONTROL\_ | AurLeEnDtn\_GbNro\_TTEMP consecutive failed connection attempts per account | TS |  |
| [ENABLED\_ROLES](#_bookmark84) | Roles enabled within current session | 8.0.19 |  |
| [ENGINES](#_bookmark85) | Storage engine  properties |  |  |
| [EVENTS](#_bookmark53) | Event Manager events |  |  |
| [FILES](#_bookmark86) | Files that store  tablespace data |  |  |
| [INNODB\_BUFFER\_PAGE](#_bookmark120) | Pages in InnoDB buffer pool |  |  |
| INNODB\_BUFFER\_PAGE\_ | LRU ordering of pages in InnoDB buffer pool |  |  |
| INNODB\_BUFFER\_POOL\_ | SInTnAoTDSB buffer pool  statistics |  |  |
| INNODB\_CACHED\_INDEX | Sumber of index pages cached per index in InnoDB buffer pool |  |  |
| [INNODB\_CMP](#_bookmark124) | Status for operations related to compressed InnoDB tables |  |  |
| [INNODB\_CMP\_PER\_INDE](#_bookmark126) | tatus for operations related to compressed InnoDB tables and indexes |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Table** **Name** | **Description** | **Introduced** | **Deprecated** |
| INNODB\_CMP\_PER\_INDE | taRt\_oTr operations related to compressed InnoDB tables and indexes |  |  |
| [INNODB\_CMP\_RESET](#_bookmark124) | Status for operations related to compressed InnoDB tables |  |  |
| [INNODB\_CMPMEM](#_bookmark125) | Status for compressed pages within InnoDB buffer pool |  |  |
| [INNODB\_CMPMEM\_RESET](#_bookmark125) | Status for compressed pages within InnoDB buffer pool |  |  |
| [INNODB\_COLUMNS](#_bookmark127) | Columns in each  InnoDB table |  |  |
| [INNODB\_DATAFILES](#_bookmark128) | Data file path information for InnoDB file-per-table and general tablespaces |  |  |
| [INNODB\_FIELDS](#_bookmark129) | Key columns of InnoDB indexes |  |  |
| [INNODB\_FOREIGN](#_bookmark130) | InnoDB foreign-key  metadata |  |  |
| [INNODB\_FOREIGN\_COLS](#_bookmark131) | InnoDB foreign-  key column status  information |  |  |
| INNODB\_FT\_BEING\_DEL | shot of INNODB\_FT\_DELETED table |  |  |
| [INNODB\_FT\_CONFIG](#_bookmark133) | Metadata for InnoDB table FULLTEXT index and associated processing |  |  |
| INNODB\_FT\_DEFAULT\_S | OeaWuOltRlDist of stopwords for InnoDB FULLTEXT indexes |  |  |
| [INNODB\_FT\_DELETED](#_bookmark135) | Rows deleted  from InnoDB table  FULLTEXT index |  |  |
| INNODB\_FT\_INDEX\_CAC | ken information for newly inserted rows in InnoDB FULLTEXT index |  |  |
| INNODB\_FT\_INDEX\_TAB | LInEverted index  information for  processing text  searches against  InnoDB table  FULLTEXT index |  |  |
| [INNODB\_INDEXES](#_bookmark138) | InnoDB index metadata |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Table** **Name** | **Description** | **Introduced** | **Deprecated** |
| [INNODB\_METRICS](#_bookmark139) | InnoDB performance information |  |  |
| INNODB\_SESSION\_TEMP | S\_iCpary- tablespace metadata | 8.0.13 |  |
| [INNODB\_TABLES](#_bookmark141) | InnoDB table metadata |  |  |
| [INNODB\_TABLESPACES](#_bookmark142) | InnoDB file-per-table, general, and undo tablespace metadata |  |  |
| INNODB\_TABLESPACES\_ | BifEfFile-per-table,  general, undo, and  system tablespace  metadata |  |  |
| [INNODB\_TABLESTATS](#_bookmark144) | InnoDB table low-level status information |  |  |
| INNODB\_TEMP\_TABLE\_I | NInrmation about active user-created InnoDB temporary tables |  |  |
| [INNODB\_TRX](#_bookmark146) | Active InnoDB  transaction information |  |  |
| [INNODB\_VIRTUAL](#_bookmark147) | InnoDB virtual  generated column  metadata |  |  |
| [KEY\_COLUMN\_USAGE](#_bookmark87) | Which key columns  have constraints |  |  |
| [KEYWORDS](#_bookmark88) | MySQL keywords |  |  |
| [MYSQL\_FIREWALL\_USER](#_bookmark158) | irewall in-memory data for account profiles |  | 8.0.26 |
| MYSQL\_FIREWALL\_WHIT | Lirll in-memory data for account profile allowlists |  | 8.0.26 |
| ndb\_transid\_mysql\_c | nDnBeranct\_mioanp  information |  |  |
| [OPTIMIZER\_TRACE](#_bookmark90) | Information produced by optimizer trace activity |  |  |
| [PARAMETERS](#_bookmark91) | Stored routine parameters and stored function return values |  |  |
| [PARTITIONS](#_bookmark3) | Table partition  information |  |  |
| [PLUGINS](#_bookmark92) | Plugin information |  |  |
| [PROCESSLIST](#_bookmark93) | Information about  currently executing  threads |  |  |
| [PROFILING](#_bookmark94) | Statement profiling  information |  |  |
| REFERENTIAL\_CONSTRA | eSign key information |  |  |
| [RESOURCE\_GROUPS](#_bookmark96) | Resource group  information |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Table** **Name** | **Description** | **Introduced** | **Deprecated** |
| [ROLE\_COLUMN\_GRANTS](#_bookmark97) | Column privileges for roles available to or granted by currently enabled roles | 8.0.19 |  |
| [ROLE\_ROUTINE\_GRANTS](#_bookmark98) | Routine privileges for roles available to or granted by currently enabled roles | 8.0.19 |  |
| [ROLE\_TABLE\_GRANTS](#_bookmark99) | Table privileges for roles available to or granted by currently enabled roles | 8.0.19 |  |
| [ROUTINES](#_bookmark50) | Stored routine  information |  |  |
| [SCHEMA\_PRIVILEGES](#_bookmark102) | Privileges defined on schemas |  |  |
| [SCHEMATA](#_bookmark100) | Schema information |  |  |
| [SCHEMATA\_EXTENSIONS](#_bookmark101) | Schema options | 8.0.22 |  |
| [ST\_GEOMETRY\_COLUMNS](#_bookmark104) | Columns in each table that store spatial data |  |  |
| ST\_SPATIAL\_REFERENC | AEv\_iYlaSbTlMsatial  reference systems |  |  |
| [ST\_UNITS\_OF\_MEASURE](#_bookmark106) | Acceptable units for  ST\_Distance() | 8.0.14 |  |
| [STATISTICS](#_bookmark103) | Table index statistics |  |  |
| [TABLE\_CONSTRAINTS](#_bookmark111) | Which tables have  constraints |  |  |
| TABLE\_CONSTRAINTS\_E | lNeScIoOnNsraint attributes for primary and secondary storage engines | 8.0.21 |  |
| [TABLE\_PRIVILEGES](#_bookmark113) | Privileges defined on tables |  |  |
| [TABLES](#_bookmark107) | Table information |  |  |
| [TABLES\_EXTENSIONS](#_bookmark108) | Table attributes for primary and secondary storage engines | 8.0.21 |  |
| [TABLESPACES](#_bookmark109) | Tablespace information |  |  |
| TABLESPACES\_EXTENSI | lespace attributes for primary storage engines | 8.0.21 |  |
| TP\_THREAD\_GROUP\_STA | Tread pool thread  group states |  |  |
| TP\_THREAD\_GROUP\_STA | Tread pool thread  group statistics |  |  |
| [TP\_THREAD\_STATE](#_bookmark152) | Thread pool thread  information |  |  |
| [TRIGGERS](#_bookmark52) | Trigger information |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Table** **Name** | **Description** | **Introduced** | **Deprecated** |
| [USER\_ATTRIBUTES](#_bookmark114) | User comments and attributes | 8.0.21 |  |
| [USER\_PRIVILEGES](#_bookmark115) | Privileges defined  globally per user |  |  |
| [VIEW\_ROUTINE\_USAGE](#_bookmark116) | Stored functions used in views | 8.0.13 |  |
| [VIEW\_TABLE\_USAGE](#_bookmark117) | Tables and views used in views | 8.0.13 |  |
| [VIEWS](#_bookmark54) | View information |  |  |

**26.3** **INFORMATION\_SCHEMA** **General** **Tables**

The following sections describe what may be denoted as the “general” set of INFORMATION\_SCHEMA tables. These are the tables not associated with particular storage engines, components, or plugins.

**26.3.1** **INFORMATION\_SCHEMA** **General** **Table** **Reference**

The following table summarizes INFORMATION\_SCHEMA general tables. For greater detail, see the individual table descriptions.

**Table** **26.2** **INFORMATION\_SCHEMA** **General** **Tables**

|  |  |  |
| --- | --- | --- |
| **Table** **Name** | **Description** | **Introduced** |
| ADMINISTRABLE\_ROLE\_AUTHOR | IGZrAaaObNleSusers or roles for current user or role | 8.0.19 |
| [APPLICABLE\_ROLES](#_bookmark75) | Applicable roles for current user | 8.0.19 |
| [CHARACTER\_SETS](#_bookmark76) | Available character sets |  |
| [CHECK\_CONSTRAINTS](#_bookmark77) | Table and column CHECK  constraints | 8.0.16 |
| COLLATION\_CHARACTER\_SET\_A | PCPhLaIrtBetTpplicable to each collation |  |
| [COLLATIONS](#_bookmark78) | Collations for each character set |  |
| [COLUMN\_PRIVILEGES](#_bookmark82) | Privileges defined on columns |  |
| [COLUMN\_STATISTICS](#_bookmark83) | Histogram statistics for column values |  |
| [COLUMNS](#_bookmark80) | Columns in each table |  |
| [COLUMNS\_EXTENSIONS](#_bookmark81) | Column attributes for primary and secondary storage engines | 8.0.21 |
| [ENABLED\_ROLES](#_bookmark84) | Roles enabled within current session | 8.0.19 |
| [ENGINES](#_bookmark85) | Storage engine properties |  |
| [EVENTS](#_bookmark53) | Event Manager events |  |
| [FILES](#_bookmark86) | Files that store tablespace data |  |
| [KEY\_COLUMN\_USAGE](#_bookmark87) | Which key columns have  constraints |  |
| [KEYWORDS](#_bookmark88) | MySQL keywords |  |
| ndb\_transid\_mysql\_connect | iNoDnB\_mtraapnsaction information |  |
| [OPTIMIZER\_TRACE](#_bookmark90) | Information produced by  optimizer trace activity |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Table** **Name** | **Description** | | **Introduced** |
| [PARAMETERS](#_bookmark91) | Stored routine parameters and stored function return values | |  |
| [PARTITIONS](#_bookmark3) | Table partition information | |  |
| [PLUGINS](#_bookmark92) | Plugin information | |  |
| [PROCESSLIST](#_bookmark93) | Information about currently  executing threads | |  |
| [PROFILING](#_bookmark94) | Statement profiling information | |  |
| [REFERENTIAL\_CONSTRAINTS](#_bookmark95) | Foreign key information | |  |
| [RESOURCE\_GROUPS](#_bookmark96) | Resource group information | |  |
| [ROLE\_COLUMN\_GRANTS](#_bookmark97) | Column privileges for roles  available to or granted by  currently enabled roles | | 8.0.19 |
| [ROLE\_ROUTINE\_GRANTS](#_bookmark98) | Routine privileges for roles  available to or granted by  currently enabled roles | | 8.0.19 |
| [ROLE\_TABLE\_GRANTS](#_bookmark99) | Table privileges for roles  available to or granted by  currently enabled roles | | 8.0.19 |
| [ROUTINES](#_bookmark50) | Stored routine information | |  |
| [SCHEMA\_PRIVILEGES](#_bookmark102) | Privileges defined on schemas | |  |
| [SCHEMATA](#_bookmark100) | Schema information | |  |
| [SCHEMATA\_EXTENSIONS](#_bookmark101) | Schema options | | 8.0.22 |
| [ST\_GEOMETRY\_COLUMNS](#_bookmark104) | Columns in each table that store spatial data | |  |
| ST\_SPATIAL\_REFERENCE\_SYST | EAMvSailable spatial reference  systems | |  |
| [ST\_UNITS\_OF\_MEASURE](#_bookmark106) | Acceptable units for  ST\_Distance() | | 8.0.14 |
| [STATISTICS](#_bookmark103) | Table index statistics | |  |
| [TABLE\_CONSTRAINTS](#_bookmark111) | Which tables have constraints | |  |
| TABLE\_CONSTRAINTS\_EXTENSI | OTNaSble constraint attributes for primary and secondary storage engines | | 8.0.21 |
| [TABLE\_PRIVILEGES](#_bookmark113) | Privileges defined on tables | |  |
| [TABLES](#_bookmark107) | Table information | |  |
| [TABLES\_EXTENSIONS](#_bookmark108) | Table attributes for primary and secondary storage engines | | 8.0.21 |
| [TABLESPACES](#_bookmark109) | Tablespace information | |  |
| [TABLESPACES\_EXTENSIONS](#_bookmark110) | Tablespace attributes for storage engines | primary | 8.0.21 |
| [TRIGGERS](#_bookmark52) | Trigger information | |  |
| [USER\_ATTRIBUTES](#_bookmark114) | User comments and attributes | | 8.0.21 |
| [USER\_PRIVILEGES](#_bookmark115) | Privileges defined globally per user | |  |
| [VIEW\_ROUTINE\_USAGE](#_bookmark116) | Stored functions used in views | | 8.0.13 |

|  |  |  |
| --- | --- | --- |
| **Table** **Name** | **Description** | **Introduced** |
| [VIEW\_TABLE\_USAGE](#_bookmark117) | Tables and views used in views | 8.0.13 |
| [VIEWS](#_bookmark54) | View information |  |

**26.3.2** **The** **INFORMATION\_SCHEMA**

**ADMINISTRABLE\_ROLE\_AUTHORIZATIONS** **Table**

The [ADMINISTRABLE\_ROLE\_AUTHORIZATIONS](#_bookmark74) table (available as of MySQL 8.0.19) provides information about which roles applicable for the current user or role can be granted to other users or roles.

The [ADMINISTRABLE\_ROLE\_AUTHORIZATIONS](#_bookmark74) table has these columns:

• USER

The user name part of the current user account.

• HOST

The host name part of the current user account.

• GRANTEE

The user name part of the account to which the role is granted.

• GRANTEE\_HOST

The host name part of the account to which the role is granted.

• ROLE\_NAME

The user name part of the granted role.

• ROLE\_HOST

The host name part of the granted role.

• IS\_GRANTABLE

YES or NO, depending on whether the role is grantable to other accounts.

• IS\_DEFAULT

YES or NO, depending on whether the role is a default role.

• IS\_MANDATORY

YES or NO, depending on whether the role is mandatory.

**26.3.3** **The** **INFORMATION\_SCHEMA** **APPLICABLE\_ROLES** **Table**

The [APPLICABLE\_ROLES](#_bookmark75) table (available as of MySQL 8.0.19) provides information about the roles that are applicable for the current user.

The [APPLICABLE\_ROLES](#_bookmark75) table has these columns:

• USER

The user name part of the current user account.

• HOST

The host name part of the current user account.

• GRANTEE

The user name part of the account to which the role is granted.

• GRANTEE\_HOST

The host name part of the account to which the role is granted.

• ROLE\_NAME

The user name part of the granted role.

• ROLE\_HOST

The host name part of the granted role.

• IS\_GRANTABLE

YES or NO, depending on whether the role is grantable to other accounts.

• IS\_DEFAULT

YES or NO, depending on whether the role is a default role.

• IS\_MANDATORY

YES or NO, depending on whether the role is mandatory.

**26.3.4** **The** **INFORMATION\_SCHEMA** **CHARACTER\_SETS** **Table**

The [CHARACTER\_SETS](#_bookmark76) table provides information about available character sets. The [CHARACTER\_SETS](#_bookmark76) table has these columns:

• CHARACTER\_SET\_NAME The character set name.

• DEFAULT\_COLLATE\_NAME

The default collation for the character set.

• DESCRIPTION

A description of the character set.

• MAXLEN

The maximum number of bytes required to store one character.

**Notes**

Character set information is also available from the SHOW CHARACTER SET statement. See Section 13.7.7.3, “SHOW CHARACTER SET Statement” . The following statements are equivalent:

SELECT \* FROM INFORMATION\_SCHEMA.CHARACTER\_SETS

[WHERE CHARACTER\_SET\_NAME LIKE '*wild*']

SHOW CHARACTER SET

[LIKE '*wild*']

**26.3.5** **The** **INFORMATION\_SCHEMA** **CHECK\_CONSTRAINTS** **Table**

As of MySQL 8.0.16, CREATE TABLE permits the core features of table and column CHECK constraints, and the [CHECK\_CONSTRAINTS](#_bookmark77) table provides information about these constraints.

The [CHECK\_CONSTRAINTS](#_bookmark77) table has these columns:

• CONSTRAINT\_CATALOG

The name of the catalog to which the constraint belongs. This value is always def.

• CONSTRAINT\_SCHEMA

The name of the schema (database) to which the constraint belongs.

• CONSTRAINT\_NAME The name of the constraint.

• CHECK\_CLAUSE

The expression that specifies the constraint condition.

**26.3.6** **The** **INFORMATION\_SCHEMA** **COLLATIONS** **Table**

The [COLLATIONS](#_bookmark78) table provides information about collations for each character set. The [COLLATIONS](#_bookmark78) table has these columns:

• COLLATION\_NAME The collation name.

• CHARACTER\_SET\_NAME

The name of the character set with which the collation is associated.

• ID

The collation ID.

• IS\_DEFAULT

Whether the collation is the default for its character set.

• IS\_COMPILED

Whether the character set is compiled into the server.

• SORTLEN

This is related to the amount of memory required to sort strings expressed in the character set.

• PAD\_ATTRIBUTE

The collation pad attribute, either NO PAD or PAD SPACE. This attribute affects whether trailing spaces are significant in string comparisons; see Trailing Space Handling in Comparisons.

**Notes**

Collation information is also available from the SHOW COLLATION statement. See Section 13.7.7.4, “SHOW COLLATION Statement” . The following statements are equivalent:

SELECT COLLATION\_NAME FROM INFORMATION\_SCHEMA.COLLATIONS

[WHERE COLLATION\_NAME LIKE '*wild*']

SHOW COLLATION

[LIKE '*wild*']

**26.3.7** **The** **INFORMATION\_SCHEMA**

**COLLATION\_CHARACTER\_SET\_APPLICABILITY** **Table**

The [COLLATION\_CHARACTER\_SET\_APPLICABILITY](#_bookmark79) table indicates what character set is applicable for what collation.

The [COLLATION\_CHARACTER\_SET\_APPLICABILITY](#_bookmark79) table has these columns:

• COLLATION\_NAME The collation name.

• CHARACTER\_SET\_NAME

The name of the character set with which the collation is associated.

**Notes**

The [COLLATION\_CHARACTER\_SET\_APPLICABILITY](#_bookmark79) columns are equivalent to the first two columns displayed by the SHOW COLLATION statement.

**26.3.8** **The** **INFORMATION\_SCHEMA** **COLUMNS** **Table**

The [COLUMNS](#_bookmark80) table provides information about columns in tables. The related [ST\_GEOMETRY\_COLUMNS](#_bookmark104) table provides information about table columns that store spatial data. See [Section 26.3.35, “The INFORMATION\_SCHEMA ST\_GEOMETRY\_COLUMNS Table”](#_bookmark104) .

The [COLUMNS](#_bookmark80) table has these columns:

• TABLE\_CATALOG

The name of the catalog to which the table containing the column belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the table containing the column belongs.

• TABLE\_NAME

The name of the table containing the column.

• COLUMN\_NAME The name of the column.

• ORDINAL\_POSITION

The position of the column within the table. ORDINAL\_POSITION is necessary because you might want to say ORDER BY ORDINAL\_POSITION. Unlike SHOW COLUMNS, SELECT from the [COLUMNS](#_bookmark80) table does not have automatic ordering.

• COLUMN\_DEFAULT

The default value for the column. This is NULL if the column has an explicit default of NULL, or if the column definition includes no DEFAULT clause.

• IS\_NULLABLE

The column nullability. The value is YES if NULL values can be stored in the column, NO if not.

• DATA\_TYPE

The column data type.

The DATA\_TYPE value is the type name only with no other information. The COLUMN\_TYPE value contains the type name and possibly other information such as the precision or length.

• CHARACTER\_MAXIMUM\_LENGTH

For string columns, the maximum length in characters.

• CHARACTER\_OCTET\_LENGTH

For string columns, the maximum length in bytes.

• NUMERIC\_PRECISION

For numeric columns, the numeric precision.

• NUMERIC\_SCALE

For numeric columns, the numeric scale.

• DATETIME\_PRECISION

For temporal columns, the fractional seconds precision.

• CHARACTER\_SET\_NAME

For character string columns, the character set name.

• COLLATION\_NAME

For character string columns, the collation name.

• COLUMN\_TYPE The column data type.

The DATA\_TYPE value is the type name only with no other information. The COLUMN\_TYPE value contains the type name and possibly other information such as the precision or length.

• COLUMN\_KEY

Whether the column is indexed:

• If COLUMN\_KEY is empty, the column either is not indexed or is indexed only as a secondary column in a multiple-column, nonunique index.

• If COLUMN\_KEY is PRI, the column is a PRIMARY KEY or is one of the columns in a multiple- column PRIMARY KEY.

• If COLUMN\_KEY is UNI, the column is the first column of a UNIQUE index. (A UNIQUE index permits multiple NULL values, but you can tell whether the column permits NULL by checking the Null column.)

• If COLUMN\_KEY is MUL, the column is the first column of a nonunique index in which multiple occurrences of a given value are permitted within the column.

If more than one of the COLUMN\_KEY values applies to a given column of a table, COLUMN\_KEY displays the one with the highest priority, in the order PRI, UNI, MUL.

A UNIQUE index may be displayed as PRI if it cannot contain NULL values and there is no PRIMARY KEY in the table. A UNIQUE index may display as MUL if several columns form a composite UNIQUE

index; although the combination of the columns is unique, each column can still hold multiple occurrences of a given value.

• EXTRA

Any additional information that is available about a given column. The value is nonempty in these cases:

• auto\_increment for columns that have the AUTO\_INCREMENT attribute.

• on update CURRENT\_TIMESTAMP for TIMESTAMP or DATETIME columns that have the ON UPDATE CURRENT\_TIMESTAMP attribute.

• STORED GENERATED or VIRTUAL GENERATED for generated columns.

• DEFAULT\_GENERATED for columns that have an expression default value.

• PRIVILEGES

The privileges you have for the column.

• COLUMN\_COMMENT

Any comment included in the column definition.

• GENERATION\_EXPRESSION

For generated columns, displays the expression used to compute column values. Empty for nongenerated columns. For information about generated columns, see Section 13.1.20.8, “CREATE TABLE and Generated Columns” .

• SRS\_ID

This value applies to spatial columns. It contains the column SRID value that indicates the spatial reference system for values stored in the column. See Section 11.4.1, “Spatial Data Types” , and Section 11.4.5, “Spatial Reference System Support” . The value is NULL for nonspatial columns and spatial columns with no SRID attribute.

**Notes**

• In SHOW COLUMNS, the Type display includes values from several different [COLUMNS](#_bookmark80) columns.

• CHARACTER\_OCTET\_LENGTH should be the same as CHARACTER\_MAXIMUM\_LENGTH, except for multibyte character sets.

• CHARACTER\_SET\_NAME can be derived from COLLATION\_NAME. For example, if you say SHOW FULL COLUMNS FROM t, and you see in the COLLATION\_NAME column a value of

utf8mb4\_swedish\_ci, the character set is what appears before the first underscore: utf8mb4.

Column information is also available from the SHOW COLUMNS statement. See Section 13.7.7.5, “SHOW COLUMNS Statement” . The following statements are nearly equivalent:

SELECT COLUMN\_NAME, DATA\_TYPE, IS\_NULLABLE, COLUMN\_DEFAULT

FROM INFORMATION\_SCHEMA.COLUMNS

WHERE table\_name = '*tbl\_name* '

[AND table\_schema = '*db\_name* ']

[AND column\_name LIKE '*wild*']

SHOW COLUMNS

FROM *tbl\_name*

[FROM *db\_name*]

[LIKE '*wild*']

In MySQL 8.0.30 and later, information about generated invisible primary key columns is visible in this table by default. You can cause such information to be hidden by setting



show\_gipk\_in\_create\_table\_and\_information\_schema = OFF. For more information, see Section 13.1.20.11, “Generated Invisible Primary Keys” .

**26.3.9** **The** **INFORMATION\_SCHEMA** **COLUMNS\_EXTENSIONS** **Table**

The COLUMNS\_EXTENSIONS table (available as of MySQL 8.0.21) provides information about column attributes defined for primary and secondary storage engines.

**Note**

The [COLUMNS\_EXTENSIONS](#_bookmark81) table is reserved for future use.

The [COLUMNS\_EXTENSIONS](#_bookmark81) table has these columns:

• TABLE\_CATALOG

The name of the catalog to which the table belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the table belongs.

• TABLE\_NAME The name of the table.

• COLUMN\_NAME The name of the column.

• ENGINE\_ATTRIBUTE

Column attributes defined for the primary storage engine. Reserved for future use.

• SECONDARY\_ENGINE\_ATTRIBUTE

Column attributes defined for the secondary storage engine. Reserved for future use.

**26.3.10** **The** **INFORMATION\_SCHEMA** **COLUMN\_PRIVILEGES** **Table**

The [COLUMN\_PRIVILEGES](#_bookmark82) table provides information about column privileges. It takes its values from the mysql.columns\_priv system table.

The [COLUMN\_PRIVILEGES](#_bookmark82) table has these columns:

• GRANTEE

The name of the account to which the privilege is granted, in '*user\_name*'@'*host\_name*' format.

• TABLE\_CATALOG

The name of the catalog to which the table containing the column belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the table containing the column belongs.

• TABLE\_NAME

The name of the table containing the column.

• COLUMN\_NAME The name of the column.

• PRIVILEGE\_TYPE

The privilege granted. The value can be any privilege that can be granted at the column level; see Section 13.7.1.6, “GRANT Statement” . Each row lists a single privilege, so there is one row per column privilege held by the grantee.

In the output from SHOW FULL COLUMNS, the privileges are all in one column and in lowercase, for example, select,insert,update,references. In [COLUMN\_PRIVILEGES](#_bookmark82), there is one privilege per row, in uppercase.

• IS\_GRANTABLE

YES if the user has the GRANT OPTION privilege, NO otherwise. The output does not list GRANT OPTION as a separate row with PRIVILEGE\_TYPE='GRANT OPTION'.

**Notes**

• [COLUMN\_PRIVILEGES](#_bookmark82) is a nonstandard INFORMATION\_SCHEMA table.

The following statements are *not* equivalent:

SELECT ... FROM INFORMATION\_SCHEMA.COLUMN\_PRIVILEGES

SHOW GRANTS ...

**26.3.11** **The** **INFORMATION\_SCHEMA** **COLUMN\_STATISTICS** **Table**

The [COLUMN\_STATISTICS](#_bookmark83) table provides access to histogram statistics for column values.

For information about histogram statistics, see Section 8.9.6, “Optimizer Statistics” , and Section 13.7.3.1, “ANALYZE TABLE Statement” .

You can see information only for columns for which you have some privilege.

The [COLUMN\_STATISTICS](#_bookmark83) table has these columns:

• SCHEMA\_NAME

The names of the schema for which the statistics apply.

• TABLE\_NAME

The names of the column for which the statistics apply.

• COLUMN\_NAME

The names of the column for which the statistics apply.

• HISTOGRAM

A JSON object describing the column statistics, stored as a histogram.

**26.3.12** **The** **INFORMATION\_SCHEMA** **ENABLED\_ROLES** **Table**

The [ENABLED\_ROLES](#_bookmark84) table (available as of MySQL 8.0. 19) provides information about the roles that

are enabled within the current session.

The [ENABLED\_ROLES](#_bookmark84) table has these columns:

• ROLE\_NAME

The user name part of the granted role.

• ROLE\_HOST

The host name part of the granted role.

• IS\_DEFAULT

YES or NO, depending on whether the role is a default role.

• IS\_MANDATORY

YES or NO, depending on whether the role is mandatory.

**26.3.13** **The** **INFORMATION\_SCHEMA** **ENGINES** **Table**

The [ENGINES](#_bookmark85) table provides information about storage engines. This is particularly useful for checking whether a storage engine is supported, or to see what the default engine is.

The [ENGINES](#_bookmark85) table has these columns:

• ENGINE

The name of the storage engine.

• SUPPORT

The server's level of support for the storage engine, as shown in the following table.

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| YES | The engine is supported and is active |
| DEFAULT | Like YES, plus this is the default engine |
| NO | The engine is not supported |
| DISABLED | The engine is supported but has been disabled |

A value of NO means that the server was compiled without support for the engine, so it cannot be enabled at runtime.

A value of DISABLED occurs either because the server was started with an option that disables the engine, or because not all options required to enable it were given. In the latter case, the error log should contain a reason indicating why the option is disabled. See Section 5.4.2, “The Error Log” .

You might also see DISABLED for a storage engine if the server was compiled to support it, but was started with a --skip-*engine\_name* option. For the NDB storage engine, DISABLED means the server was compiled with support for NDB Cluster, but was not started with the --ndbcluster option.

All MySQL servers support MyISAM tables. It is not possible to disable MyISAM.

• COMMENT

A brief description of the storage engine.

• TRANSACTIONS

Whether the storage engine supports transactions.

• XA

Whether the storage engine supports XA transactions.

• SAVEPOINTS

Whether the storage engine supports savepoints.

**Notes**

• [ENGINES](#_bookmark85) is a nonstandard INFORMATION\_SCHEMA table.

Storage engine information is also available from the SHOW ENGINES statement. See Section 13.7.7.16, “SHOW ENGINES Statement” . The following statements are equivalent:

SELECT \* FROM INFORMATION\_SCHEMA.ENGINES

SHOW ENGINES

**26.3.14** **The** **INFORMATION\_SCHEMA** **EVENTS** **Table**

The [EVENTS](#_bookmark53) table provides information about Event Manager events, which are discussed in [Section 25.4, “Using the Event Scheduler”](#_bookmark33) .

The [EVENTS](#_bookmark53) table has these columns:

• EVENT\_CATALOG

The name of the catalog to which the event belongs. This value is always def.

• EVENT\_SCHEMA

The name of the schema (database) to which the event belongs.

• EVENT\_NAME The name of the event.

• DEFINER

The account named in the DEFINER clause (often the user who created the event), in '*user\_name*'@'*host\_name*' format.

• TIME\_ZONE

The event time zone, which is the time zone used for scheduling the event and that is in effect within the event as it executes. The default value is SYSTEM.

• EVENT\_BODY

The language used for the statements in the event's DO clause. The value is always SQL.

• EVENT\_DEFINITION

The text of the SQL statement making up the event's DO clause; in other words, the statement executed by this event.

• EVENT\_TYPE

The event repetition type, either ONE TIME (transient) or RECURRING (repeating).

• EXECUTE\_AT

For a one-time event, this is the DATETIME value specified in the AT clause of the CREATE EVENT statement used to create the event, or of the last ALTER EVENT statement that modified the event. The value shown in this column reflects the addition or subtraction of any INTERVAL value included in the event's AT clause. For example, if an event is created using ON SCHEDULE AT CURRENT\_TIMESTAMP + '1:6' DAY\_HOUR, and the event was created at 2018-02-09 14:05:30, the value shown in this column would be '2018-02-10 20:05:30'. If the event's timing is determined by an EVERY clause instead of an AT clause (that is, if the event is recurring), the value of this column is NULL.

• INTERVAL\_VALUE

For a recurring event, the number of intervals to wait between event executions. For a transient event, the value is always NULL.

• INTERVAL\_FIELD

The time units used for the interval which a recurring event waits before repeating. For a transient event, the value is always NULL.

• SQL\_MODE

The SQL mode in effect when the event was created or altered, and under which the event executes. For the permitted values, see Section 5.1.11, “Server SQL Modes” .

• STARTS

The start date and time for a recurring event. This is displayed as a DATETIME value, and is NULL if no start date and time are defined for the event. For a transient event, this column is always NULL. For a recurring event whose definition includes a STARTS clause, this column contains the corresponding DATETIME value. As with the EXECUTE\_AT column, this value resolves any expressions used. If there is no STARTS clause affecting the timing of the event, this column is NULL

• ENDS

For a recurring event whose definition includes a ENDS clause, this column contains the corresponding DATETIME value. As with the EXECUTE\_AT column, this value resolves any expressions used. If there is no ENDS clause affecting the timing of the event, this column is NULL.

• STATUS

The event status. One of ENABLED, DISABLED, or SLAVESIDE\_DISABLED. SLAVESIDE\_DISABLED indicates that the creation of the event occurred on another MySQL server acting as a replication source and replicated to the current MySQL server which is acting as a replica, but the event is not presently being executed on the replica. For more information, see Section 17.5.1.16, “Replication of Invoked Features” . information.

• ON\_COMPLETION

One of the two values PRESERVE or NOT PRESERVE.

• CREATED

The date and time when the event was created. This is a TIMESTAMP value.

• LAST\_ALTERED

The date and time when the event was last modified. This is a TIMESTAMP value. If the event has not been modified since its creation, this value is the same as the CREATED value.

• LAST\_EXECUTED

The date and time when the event last executed. This is a DATETIME value. If the event has never executed, this column is NULL.

LAST\_EXECUTED indicates when the event started. As a result, the ENDS column is never less than

LAST\_EXECUTED.

• EVENT\_COMMENT

The text of the comment, if the event has one. If not, this value is empty.

• ORIGINATOR

The server ID of the MySQL server on which the event was created; used in replication. This value may be updated by ALTER EVENT to the server ID of the server on which that statement occurs, if executed on a replication source. The default value is 0.

• CHARACTER\_SET\_CLIENT

The session value of the character\_set\_client system variable when the event was created.

• COLLATION\_CONNECTION

The session value of the collation\_connection system variable when the event was created.

• DATABASE\_COLLATION

The collation of the database with which the event is associated.

**Notes**

• [EVENTS](#_bookmark53) is a nonstandard INFORMATION\_SCHEMA table.

• Times in the [EVENTS](#_bookmark53) table are displayed using the event time zone, the current session time zone, or UTC, as described in [Section 25.4.4, “Event Metadata”](#_bookmark37) .

• For more information about SLAVESIDE\_DISABLED and the ORIGINATOR column, see Section 17.5.1.16, “Replication of Invoked Features” .

**Example**

Suppose that the user 'jon'@'ghidora' creates an event named e\_daily, and then modifies it a few minutes later using an ALTER EVENT statement, 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 ;

ALTER EVENT e\_daily

ENABLE;

(Note that comments can span multiple lines.)

This user can then run the following SELECT statement, and obtain the output shown:

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.EVENTS**

**WHERE** **EVENT\_NAME** **=** **'e\_daily'**

**AND** **EVENT\_SCHEMA** **=** **'myschema'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

EVENT\_CATALOG: def

EVENT\_SCHEMA: myschema

EVENT\_NAME: e\_daily

DEFINER: jon@ghidora

TIME\_ZONE: SYSTEM

EVENT\_BODY: SQL

EVENT\_DEFINITION: BEGIN

INSERT INTO site\_activity.totals (time, total)

SELECT CURRENT\_TIMESTAMP, COUNT(\*)

FROM site\_activity.sessions;

DELETE FROM site\_activity.sessions;

END

EVENT\_TYPE: RECURRING

EXECUTE\_AT: NULL

INTERVAL\_VALUE: 1

INTERVAL\_FIELD: DAY

SQL\_MODE: ONLY\_FULL\_GROUP\_BY,STRICT\_TRANS\_TABLES,

NO\_ZERO\_IN\_DATE,NO\_ZERO\_DATE,

ERROR\_FOR\_DIVISION\_BY\_ZERO,

NO\_ENGINE\_SUBSTITUTION

STARTS: 2018-08-08 11:06:34

ENDS: NULL

STATUS: ENABLED

ON\_COMPLETION: NOT PRESERVE

CREATED: 2018-08-08 11:06:34

LAST\_ALTERED: 2018-08-08 11:06:34

LAST\_EXECUTED: 2018-08-08 16:06:34

EVENT\_COMMENT: Saves total number of sessions then clears the

table each day

ORIGINATOR: 1

CHARACTER\_SET\_CLIENT: utf8mb4

COLLATION\_CONNECTION: utf8mb4\_0900\_ai\_ci

DATABASE\_COLLATION: utf8mb4\_0900\_ai\_ci

Event information is also available from the SHOW EVENTS statement. See Section 13.7.7.18, “SHOW EVENTS Statement” . The following statements are equivalent:

SELECT

EVENT\_SCHEMA, EVENT\_NAME, DEFINER, TIME\_ZONE, EVENT\_TYPE, EXECUTE\_AT,

INTERVAL\_VALUE, INTERVAL\_FIELD, STARTS, ENDS, STATUS, ORIGINATOR,

CHARACTER\_SET\_CLIENT, COLLATION\_CONNECTION, DATABASE\_COLLATION

FROM INFORMATION\_SCHEMA.EVENTS

WHERE table\_schema = '*db\_name* '

[AND column\_name LIKE '*wild*']

SHOW EVENTS

[FROM *db\_name*]

[LIKE '*wild*']

**26.3.15** **The** **INFORMATION\_SCHEMA** **FILES** **Table**

The FILES table provides information about the files in which MySQL tablespace data is stored.

The FILES table provides information about InnoDB data files. In NDB Cluster, this table also provides information about the files in which NDB Cluster Disk Data tables are stored. For additional information specific to InnoDB, see [InnoDB Notes](#_bookmark169), later in this section; for additional information specific to NDB Cluster, see [NDB Notes](#_bookmark170).

The FILES table has these columns:

• FILE\_ID

For InnoDB: The tablespace ID, also referred to as the space\_id or fil\_space\_t::id. For NDB: A file identifier. FILE\_ID column values are auto-generated.

• FILE\_NAME

For InnoDB: The name of the data file. File-per-table and general tablespaces have an .ibd file name extension. Undo tablespaces are prefixed by undo. The system tablespace is prefixed by ibdata. The global temporary tablespace is prefixed by ibtmp. The file name includes the file path, which may be relative to the MySQL data directory (the value of the datadir system variable).

For NDB: The name of an undo log file created by CREATE LOGFILE GROUP or ALTER LOGFILE GROUP, or of a data file created by CREATE TABLESPACE or ALTER TABLESPACE. In NDB

8.0, the file name is shown with a relative path; for an undo log file, this path is relative to the

directory DataDir/ndb\_NodeId\_fs/LG; for a data file, it is relative to the directory DataDir/ ndb\_NodeId\_fs/TS. This means, for example, that the name of a data file created with ALTER TABLESPACE ts ADD DATAFILE 'data\_2.dat' INITIAL SIZE 256M is shown as ./ data\_2.dat.

• FILE\_TYPE

For InnoDB: The tablespace file type. There are three possible file types for InnoDB files. TABLESPACE is the file type for any system, general, or file-per-table tablespace file that holds tables, indexes, or other forms of user data. TEMPORARY is the file type for temporary tablespaces. UNDO LOG is the file type for undo tablespaces, which hold undo records.

For NDB: One of the values UNDO LOG or DATAFILE. Prior to NDB 8.0.13, TABLESPACE was also a possible value.

• TABLESPACE\_NAME

The name of the tablespace with which the file is associated.

For InnoDB: General tablespace names are as specified when created. File-per-table tablespace names are shown in the following format: *schema\_name*/*table\_name*. The InnoDB system tablespace name is innodb\_system. The global temporary tablespace name is innodb\_temporary. Default undo tablespace names are innodb\_undo\_001 and innodb\_undo\_002. User-created undo tablespace names are as specified when created.

• TABLE\_CATALOG

This value is always empty.

• TABLE\_SCHEMA

This is always NULL.

• TABLE\_NAME

This is always NULL.

• LOGFILE\_GROUP\_NAME For InnoDB: This is always NULL.

For NDB: The name of the log file group to which the log file or data file belongs.

• LOGFILE\_GROUP\_NUMBER For InnoDB: This is always NULL.

For NDB: For a Disk Data undo log file, the auto-generated ID number of the log file group to which the log file belongs. This is the same as the value shown for the id column in the ndbinfo.dict\_obj\_info table and the log\_id column in the ndbinfo.logspaces and ndbinfo.logspaces tables for this undo log file.

• ENGINE

For InnoDB: This value is always InnoDB.

For NDB: This value is always ndbcluster.

• FULLTEXT\_KEYS This is always NULL.

• DELETED\_ROWS

This is always NULL.

• UPDATE\_COUNT

This is always NULL.

• FREE\_EXTENTS

For InnoDB: The number of fully free extents in the current data file.

For NDB: The number of extents which have not yet been used by the file.

• TOTAL\_EXTENTS

For InnoDB: The number of full extents used in the current data file. Any partial extent at the end of the file is not counted.

For NDB: The total number of extents allocated to the file.

• EXTENT\_SIZE

For InnoDB: Extent size is 1048576 (1MB) for files with a 4KB, 8KB, or 16KB page size. Extent size is 2097152 bytes (2MB) for files with a 32KB page size, and 4194304 (4MB) for files with a 64KB page size. [FILES](#_bookmark86) does not report InnoDB page size. Page size is defined by the innodb\_page\_size system variable. Extent size information can also be retrieved from the [INNODB\_TABLESPACES](#_bookmark142) table where FILES.FILE\_ID = INNODB\_TABLESPACES.SPACE.

For NDB: The size of an extent for the file in bytes.

• INITIAL\_SIZE

For InnoDB: The initial size of the file in bytes.

For NDB: The size of the file in bytes. This is the same value that was used in the INITIAL\_SIZE clause of the CREATE LOGFILE GROUP, ALTER LOGFILE GROUP, CREATE TABLESPACE, or ALTER TABLESPACE statement used to create the file.

• MAXIMUM\_SIZE

For InnoDB: The maximum number of bytes permitted in the file. The value is NULL for all data files except for predefined system tablespace data files. Maximum system tablespace file size is defined by innodb\_data\_file\_path. Maximum global temporary tablespace file size is defined by innodb\_temp\_data\_file\_path. A NULL value for a predefined system tablespace data file indicates that a file size limit was not defined explicitly.

For NDB: This value is always the same as the INITIAL\_SIZE value.

• AUTOEXTEND\_SIZE

The auto-extend size of the tablespace. For NDB, AUTOEXTEND\_SIZE is always NULL.

• CREATION\_TIME This is always NULL.

• LAST\_UPDATE\_TIME This is always NULL.

• LAST\_ACCESS\_TIME This is always NULL.

• RECOVER\_TIME

This is always NULL.

• TRANSACTION\_COUNTER This is always NULL.

• VERSION

For InnoDB: This is always NULL.

For NDB: The version number of the file.

• ROW\_FORMAT

For InnoDB: This is always NULL.

For NDB: One of FIXED or DYNAMIC.

• TABLE\_ROWS

This is always NULL.

• AVG\_ROW\_LENGTH This is always NULL.

• DATA\_LENGTH

This is always NULL.

• MAX\_DATA\_LENGTH This is always NULL.

• INDEX\_LENGTH

This is always NULL.

• DATA\_FREE

For InnoDB: The total amount of free space (in bytes) for the entire tablespace. Predefined system tablespaces, which include the system tablespace and temporary table tablespaces, may have one or more data files.

For NDB: This is always NULL.

• CREATE\_TIME

This is always NULL.

• UPDATE\_TIME

This is always NULL.

• CHECK\_TIME

This is always NULL.

• CHECKSUM

This is always NULL.

• STATUS

For InnoDB: This value is NORMAL by default. InnoDB file-per-table tablespaces may report IMPORTING, which indicates that the tablespace is not yet available.

For NDB: For NDB Cluster Disk Data files, this value is always NORMAL.

• EXTRA

For InnoDB: This is always NULL.

For NDB: (*NDB* *8.0.15* *and* *later*) For undo log files, this column shows the undo log buffer size; for data files, it is always *NULL*. A more detailed explanation is provided in the next few paragraphs.

NDBCLUSTER stores a copy of each data file and each undo log file on each data node in the cluster. In NDB 8.0.13 and later, the FILES table contains only one row for each such file. Suppose that you run the following two statements on an NDB Cluster with four data nodes:

CREATE LOGFILE GROUP mygroup

ADD UNDOFILE 'new\_undo.dat'

INITIAL\_SIZE 2G

ENGINE NDBCLUSTER;

CREATE TABLESPACE myts

ADD DATAFILE 'data\_1 .dat'

USE LOGFILE GROUP mygroup

INITIAL\_SIZE 256M

ENGINE NDBCLUSTER;

After running these two statements successfully, you should see a result similar to the one shown here for this query against the [FILES](#_bookmark86) table:

mysql> **SELECT** **LOGFILE\_GROUP\_NAME,** **FILE\_TYPE,** **EXTRA**

|  |  |
| --- | --- |
| ->  -> | **FROM** **INFORMATION\_SCHEMA.FILES**  **WHERE** **ENGINE** **=** **'ndbcluster';** |

+--------------------+-----------+--------------------------+

| LOGFILE\_GROUP\_NAME | FILE\_TYPE | EXTRA |

+--------------------+-----------+--------------------------+

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  | | mygroup  mygroup | |  | | UNDO LOG  DATAFILE | |  | | UNDO\_BUFFER\_SIZE=8388608 NULL | |  | |

+--------------------+-----------+--------------------------+

The undo log buffer size information was inadvertently removed in NDB 8.0.13, but was restored in NDB 8.0.15. (Bug #92796, Bug #28800252)

Prior to NDB 8.0.13, the FILES table contained a row for each of these files on each data node the file belonged to, as well as the size of its undo buffer. In these versions, the result of the same query contains one row per data node, as shown here:

+--------------------+-----------+-----------------------------------------+

| LOGFILE\_GROUP\_NAME | FILE\_TYPE | EXTRA |

+--------------------+-----------+-----------------------------------------+

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

mygroup

mygroup

mygroup

mygroup

mygroup

mygroup

mygroup

mygroup

|

|

|

|

|

|

|

|

CLUSTER\_NODE=5;UNDO\_BUFFER\_SIZE=8388608

CLUSTER\_NODE=6;UNDO\_BUFFER\_SIZE=8388608

CLUSTER\_NODE=7;UNDO\_BUFFER\_SIZE=8388608

CLUSTER\_NODE=8;UNDO\_BUFFER\_SIZE=8388608

CLUSTER\_NODE=5

CLUSTER\_NODE=6

CLUSTER\_NODE=7

CLUSTER\_NODE=8

UNDO LOG

UNDO LOG

UNDO LOG

UNDO LOG

DATAFILE

DATAFILE

DATAFILE

DATAFILE

|

|

|

|

|

|

|

+--------------------+-----------+-----------------------------------------+

**Notes**

• [FILES](#_bookmark86) is a nonstandard INFORMATION\_SCHEMA table.

• As of MySQL 8.0.21, you must have the PROCESS privilege to query this table.



**InnoDB** **Notes**

The following notes apply to InnoDB data files.

• Information reported by FILES is obtained from the InnoDB in-memory cache for open files, whereas [INNODB\_DATAFILES](#_bookmark128) gets its data from the InnoDB SYS\_DATAFILES internal data dictionary table.

• The information provided by FILES includes global temporary tablespace information which is not available in the InnoDB SYS\_DATAFILES internal data dictionary table, and is therefore not included in [INNODB\_DATAFILES](#_bookmark128).

• Undo tablespace information is shown in FILES when separate undo tablespaces are present, as they are by default in MySQL 8.0.

• The following query returns all FILES table information relating to InnoDB tablespaces.

SELECT

FILE\_ID, FILE\_NAME, FILE\_TYPE, TABLESPACE\_NAME, FREE\_EXTENTS,

TOTAL\_EXTENTS, EXTENT\_SIZE, INITIAL\_SIZE, MAXIMUM\_SIZE,

AUTOEXTEND\_SIZE, DATA\_FREE, STATUS

FROM INFORMATION\_SCHEMA.FILES

WHERE ENGINE='InnoDB'\G

**NDB** **Notes**

• The FILES table provides information about Disk Data *files* only; you cannot use it for determining disk space allocation or availability for individual NDB tables. However, it is possible to see how much space is allocated for each NDB table having data stored on disk—as well as how much remains available for storage of data on disk for that table—using ndb\_desc.

• Beginning with NDB 8.0.29 much of the information in the FILES table can also be found in the ndbinfo.files table.

• The CREATION\_TIME, LAST\_UPDATE\_TIME, and LAST\_ACCESSED values are as reported by the operating system, and are not supplied by the NDB storage engine. Where no value is provided by the operating system, these columns display NULL.

• The difference between the TOTAL EXTENTS and FREE\_EXTENTS columns is the number of extents currently in use by the file:

SELECT TOTAL\_EXTENTS - FREE\_EXTENTS AS extents\_used

FROM INFORMATION\_SCHEMA.FILES

WHERE FILE\_NAME = './myfile.dat';

To approximate the amount of disk space in use by the file, multiply that difference by the value of the EXTENT\_SIZE column, which gives the size of an extent for the file in bytes:

SELECT (TOTAL\_EXTENTS - FREE\_EXTENTS) \* EXTENT\_SIZE AS bytes\_used

FROM INFORMATION\_SCHEMA.FILES

WHERE FILE\_NAME = './myfile.dat';

Similarly, you can estimate the amount of space that remains available in a given file by multiplying FREE\_EXTENTS by EXTENT\_SIZE:

SELECT FREE\_EXTENTS \* EXTENT\_SIZE AS bytes\_free

FROM INFORMATION\_SCHEMA.FILES

WHERE FILE\_NAME = './myfile.dat';

**Important**

The byte values produced by the preceding queries are approximations only, and their precision is inversely proportional to the value of EXTENT\_SIZE. That is, the larger EXTENT\_SIZE becomes, the less accurate the approximations are.

It is also important to remember that once an extent is used, it cannot be freed again without dropping the data file of which it is a part. This means that deletes from a Disk Data table do *not* release disk space.

The extent size can be set in a CREATE TABLESPACE statement. For more information, see Section 13.1.21, “CREATE TABLESPACE Statement” .

• Prior to NDB 8.0. 13, an additional row was present in the [FILES](#_bookmark86) table following the creation of a logfile group, having NULL in the FILE\_NAME column. In NDB 8.0.13 and later, this row

— which did not correspond to any file— is no longer shown, and it is necessary to query the ndbinfo.logspaces table to obtain undo log file usage information. See the description of this table as well as Section 23.6. 11. 1, “NDB Cluster Disk Data Objects” , for more information.

The remainder of the discussion in this item applies only to NDB 8.0.12 and earlier. For the row having NULL in the FILE\_NAME column, the value of the FILE\_ID column is always 0, that of the FILE\_TYPE column is always UNDO LOG, and that of the STATUS column is always NORMAL. The value of the ENGINE column is always ndbcluster.

The FREE\_EXTENTS column in this row shows the total number of free extents available to all undo files belonging to a given log file group whose name and number are shown in the LOGFILE\_GROUP\_NAME and LOGFILE\_GROUP\_NUMBER columns, respectively.

Suppose there are no existing log file groups on your NDB Cluster, and you create one using the following statement:

mysql> **CREATE** **LOGFILE** **GROUP** **lg1**

**ADD** **UNDOFILE** **'undofile.dat'**

**INITIAL\_SIZE** **=** **16M**

**UNDO\_BUFFER\_SIZE** **=** **1M**

**ENGINE** **=** **NDB;**

You can now see this NULL row when you query the [FILES](#_bookmark86) table:

mysql> **SELECT** **DISTINCT**

**FILE\_NAME** **AS** **File,**

**FREE\_EXTENTS** **AS** **Free,**

**TOTAL\_EXTENTS** **AS** **Total,**

**EXTENT\_SIZE** **AS** **Size,**

**INITIAL\_SIZE** **AS** **Initial**

**FROM** **INFORMATION\_SCHEMA.FILES;**

+--------------+---------+---------+------+----------+

| File | Free | Total | Size | Initial |

+--------------+---------+---------+------+----------+

| undofile .dat | NULL | 4194304 | 4 | 16777216 |

| NULL | 4184068 | NULL | 4 | NULL |

+--------------+---------+---------+------+----------+

The total number of free extents available for undo logging is always somewhat less than the sum of the TOTAL\_EXTENTS column values for all undo files in the log file group due to overhead required for maintaining the undo files. This can be seen by adding a second undo file to the log file group, then repeating the previous query against the [FILES](#_bookmark86) table:

mysql> **ALTER** **LOGFILE** **GROUP** **lg1**

**ADD** **UNDOFILE** **'undofile02.dat'**

**INITIAL\_SIZE** **=** **4M**

**ENGINE** **=** **NDB;**

mysql> **SELECT** **DISTINCT**

**FILE\_NAME** **AS** **File,**

**FREE\_EXTENTS** **AS** **Free,**

**TOTAL\_EXTENTS** **AS** **Total,**

**EXTENT\_SIZE** **AS** **Size,**

**INITIAL\_SIZE** **AS** **Initial**

**FROM** **INFORMATION\_SCHEMA.FILES;**

+----------------+---------+---------+------+----------+

| File | Free | Total | Size | Initial |

+----------------+---------+---------+------+----------+

| undofile .dat | NULL | 4194304 | 4 | 16777216 |

| undofile02 .dat | NULL | 1048576 | 4 | 4194304 |

| NULL | 5223944 | NULL | 4 | NULL |

+----------------+---------+---------+------+----------+

The amount of free space in bytes which is available for undo logging by Disk Data tables using this log file group can be approximated by multiplying the number of free extents by the initial size:

mysql> **SELECT**

**FREE\_EXTENTS** **AS** **'Free** **Extents',**

**FREE\_EXTENTS** **\*** **EXTENT\_SIZE** **AS** **'Free** **Bytes'**

**FROM** **INFORMATION\_SCHEMA.FILES**

**WHERE** **LOGFILE\_GROUP\_NAME** **=** **'lg1'**

**AND** **FILE\_NAME** **IS** **NULL;**

+--------------+------------+

| Free Extents | Free Bytes |

+--------------+------------+

| 5223944 | 20895776 |

+--------------+------------+

If you create an NDB Cluster Disk Data table and then insert some rows into it, you can see approximately how much space remains for undo logging afterward, for example:

mysql> **CREATE** **TABLESPACE** **ts1**

**ADD** **DATAFILE** **'data1.dat'**

**USE** **LOGFILE** **GROUP** **lg1**

**INITIAL\_SIZE** **512M**

**ENGINE** **=** **NDB;**

mysql> **CREATE** **TABLE** **dd** **(**

**c1** **INT** **NOT** **NULL** **PRIMARY** **KEY,**

**c2** **INT,**

**c3** **DATE**

**)**

**TABLESPACE** **ts1** **STORAGE** **DISK**

**ENGINE** **=** **NDB;**

mysql> **INSERT** **INTO** **dd** **VALUES**

**(NULL,** **1234567890,** **'2007-02-02'),**

**(NULL,** **1126789005,** **'2007-02-03'),**

**(NULL,** **1357924680,** **'2007-02-04'),**

**(NULL,** **1642097531,** **'2007-02-05');**

mysql> **SELECT**

**FREE\_EXTENTS** **AS** **'Free** **Extents',**

**FREE\_EXTENTS** **\*** **EXTENT\_SIZE** **AS** **'Free** **Bytes'**

**FROM** **INFORMATION\_SCHEMA.FILES**

**WHERE** **LOGFILE\_GROUP\_NAME** **=** **'lg1'**

**AND** **FILE\_NAME** **IS** **NULL;**

+--------------+------------+

| Free Extents | Free Bytes |

+--------------+------------+

| 5207565 | 20830260 |

+--------------+------------+

• Prior to NDB 8.0. 13, an additional row was present in the [FILES](#_bookmark86) table for each NDB Cluster Disk

Data tablespace. Because it did not correspond to an actual file, it was removed in NDB 8.0.13.

This row had NULL for the value of the FILE\_NAME column, the value of the FILE\_ID column was always 0, that of the FILE\_TYPE column was always TABLESPACE, that of the STATUS column was always NORMAL, and the value of the ENGINE column is always NDBCLUSTER.

In NDB 8.0.13 and later, you can obtain information about Disk Data tablespaces using the ndb\_desc utility. For more information, see Section 23.6.11.1, “NDB Cluster Disk Data Objects” , as well as the description of ndb\_desc.

• For additional information, and examples of creating, dropping, and obtaining information about NDB Cluster Disk Data objects, see Section 23.6.11, “NDB Cluster Disk Data Tables” .

**26.3.16** **The** **INFORMATION\_SCHEMA** **KEY\_COLUMN\_USAGE** **Table**

The [KEY\_COLUMN\_USAGE](#_bookmark87) table describes which key columns have constraints. This table provides no information about functional key parts because they are expressions and the table provides information only about columns.

The [KEY\_COLUMN\_USAGE](#_bookmark87) table has these columns:

• CONSTRAINT\_CATALOG

The name of the catalog to which the constraint belongs. This value is always def.

• CONSTRAINT\_SCHEMA

The name of the schema (database) to which the constraint belongs.

• CONSTRAINT\_NAME The name of the constraint.

• TABLE\_CATALOG

The name of the catalog to which the table belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the table belongs.

• TABLE\_NAME

The name of the table that has the constraint.

• COLUMN\_NAME

The name of the column that has the constraint.

If the constraint is a foreign key, then this is the column of the foreign key, not the column that the foreign key references.

• ORDINAL\_POSITION

The column's position within the constraint, not the column's position within the table. Column positions are numbered beginning with 1.

• POSITION\_IN\_UNIQUE\_CONSTRAINT

NULL for unique and primary-key constraints. For foreign-key constraints, this column is the ordinal position in key of the table that is being referenced.

• REFERENCED\_TABLE\_SCHEMA

The name of the schema referenced by the constraint.

• REFERENCED\_TABLE\_NAME

The name of the table referenced by the constraint.

• REFERENCED\_COLUMN\_NAME

The name of the column referenced by the constraint.

Suppose that there are two tables name t1 and t3 that have the following definitions:

CREATE TABLE t1

(

s1 INT,

s2 INT,

s3 INT,

PRIMARY KEY(s3)

) ENGINE=InnoDB;

CREATE TABLE t3

(

s1 INT,

s2 INT,

s3 INT,

KEY(s1),

CONSTRAINT CO FOREIGN KEY (s2) REFERENCES t1(s3)

) ENGINE=InnoDB;

For those two tables, the [KEY\_COLUMN\_USAGE](#_bookmark87) table has two rows:

• One row with CONSTRAINT\_NAME = 'PRIMARY', TABLE\_NAME = 't1', COLUMN\_NAME = 's3',

ORDINAL\_POSITION = 1, POSITION\_IN\_UNIQUE\_CONSTRAINT = NULL.

For NDB: This value is always NULL.

• One row with CONSTRAINT\_NAME = 'CO', TABLE\_NAME = 't3', COLUMN\_NAME = 's2', ORDINAL\_POSITION = 1, POSITION\_IN\_UNIQUE\_CONSTRAINT = 1.

**26.3.17** **The** **INFORMATION\_SCHEMA** **KEYWORDS** **Table**

The [KEYWORDS](#_bookmark88) table lists the words considered keywords by MySQL and, for each one, indicates whether it is reserved. Reserved keywords may require special treatment in some contexts, such as special quoting when used as identifiers (see Section 9.3, “Keywords and Reserved Words” ). This table provides applications a runtime source of MySQL keyword information.

Prior to MySQL 8.0.13, selecting from the KEYWORDS table with no default database selected produced an error. (Bug #90160, Bug #27729859)

The [KEYWORDS](#_bookmark88) table has these columns:

• WORD

The keyword.

• RESERVED

An integer indicating whether the keyword is reserved (1) or nonreserved (0).

These queries lists all keywords, all reserved keywords, and all nonreserved keywords, respectively:

SELECT \* FROM INFORMATION\_SCHEMA.KEYWORDS;

SELECT WORD FROM INFORMATION\_SCHEMA.KEYWORDS WHERE RESERVED = 1;

SELECT WORD FROM INFORMATION\_SCHEMA.KEYWORDS WHERE RESERVED = 0;

The latter two queries are equivalent to:

SELECT WORD FROM INFORMATION\_SCHEMA.KEYWORDS WHERE RESERVED;

SELECT WORD FROM INFORMATION\_SCHEMA.KEYWORDS WHERE NOT RESERVED;

If you build MySQL from source, the build process generates a keyword\_list.h header file containing an array of keywords and their reserved status. This file can be found in the sql directory under the build directory. This file may be useful for applications that require a static source for the keyword list.

**26.3.18** **The** **INFORMATION\_SCHEMA** **ndb\_transid\_mysql\_connection\_map**

**Table**

The ndb\_transid\_mysql\_connection\_map table provides a mapping between NDB transactions, NDB transaction coordinators, and MySQL Servers attached to an NDB Cluster as API nodes. This information is used when populating the server\_operations and server\_transactions tables of the ndbinfo NDB Cluster information database.

|  |  |  |
| --- | --- | --- |
| **INFORMATION\_SCHEMA** **Name** | **SHOW** **Name** | **Remarks** |
| mysql\_connection\_id |  | MySQL Server connection ID |
| node\_id |  | Transaction coordinator node ID |
| ndb\_transid |  | NDB transaction ID |

The mysql\_connection\_id is the same as the connection or session ID shown in the output of

SHOW PROCESSLIST.

There are no SHOW statements associated with this table.

This is a nonstandard table, specific to NDB Cluster. It is implemented as an INFORMATION\_SCHEMA plugin; you can verify that it is supported by checking the output of SHOW PLUGINS. If ndb\_transid\_mysql\_connection\_map support is enabled, the output from this statement includes a plugin having this name, of type INFORMATION SCHEMA, and having status ACTIVE, as shown here (using emphasized text):

mysql> **SHOW** **PLUGINS;**

+----------------------------------+--------+--------------------+---------+---------+

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| | Name | | | Status | | | Type | | Library | License | |

+----------------------------------+--------+--------------------+---------+---------+

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | binlog  | mysql\_native\_password  | sha256\_password  | caching\_sha2\_password  | sha2\_cache\_cleaner  | daemon\_keyring\_proxy\_plugin  | CSV  | MEMORY  | InnoDB  | INNODB\_TRX  | INNODB\_CMP  ...  | INNODB\_SESSION\_TEMP\_TABLESPACES  | MyISAM  | MRG\_MYISAM  | PERFORMANCE\_SCHEMA  | TempTable  | ARCHIVE  | BLACKHOLE  | ndbcluster  | ndbinfo  *|* *ndb\_transid\_mysql\_connection\_map*  | ngram  | mysqlx\_cache\_cleaner  | mysqlx | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  *|*  |  |  | | ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  ACTIVE  *ACTIVE*  ACTIVE  ACTIVE  ACTIVE | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  *|*  |  |  | | STORAGE ENGINE  AUTHENTICATION  AUTHENTICATION  AUTHENTICATION  AUDIT  DAEMON  STORAGE ENGINE  STORAGE ENGINE  STORAGE ENGINE  INFORMATION SCHEMA  INFORMATION SCHEMA  INFORMATION SCHEMA   |  |  | | --- | --- | | STORAGE  STORAGE  STORAGE  STORAGE  STORAGE  STORAGE  STORAGE  STORAGE | ENGINE  ENGINE  ENGINE  ENGINE  ENGINE  ENGINE  ENGINE  ENGINE |   *INFORMATION* *SCHEMA*  FTPARSER  AUDIT  DAEMON | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  *|*  |  |  | | NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  NULL  *NULL*  NULL  NULL  NULL | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  *|*  |  |  | | GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  GPL  *GPL*  GPL  GPL  GPL | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  *|*  |  |  | |

+----------------------------------+--------+--------------------+---------+---------+

47 rows in set (0.01 sec)

The plugin is enabled by default. You can disable it (or force the server not to run unless the plugin starts) by starting the server with the --ndb-transid-mysql-connection-map option. If the plugin is disabled, the status is shown by SHOW PLUGINS as DISABLED. The plugin cannot be enabled or disabled at runtime.

Although the names of this table and its columns are displayed using lowercase, you can use uppercase or lowercase when referring to them in SQL statements.

For this table to be created, the MySQL Server must be a binary supplied with the NDB Cluster distribution, or one built from the NDB Cluster sources with NDB storage engine support enabled. It is not available in the standard MySQL 8.0 Server.

**26.3.19** **The** **INFORMATION\_SCHEMA** **OPTIMIZER\_TRACE** **Table**

The [OPTIMIZER\_TRACE](#_bookmark90) table provides information produced by the optimizer tracing capability for traced statements. To enable tracking, use the optimizer\_trace system variable. For details, see [MySQL Internals: Tracing the Optimizer](https://dev.mysql.com/doc/internals/en/optimizer-tracing.html).

The [OPTIMIZER\_TRACE](#_bookmark90) table has these columns:

• QUERY

The text of the traced statement.

• TRACE

The trace, in JSON format.

• MISSING\_BYTES\_BEYOND\_MAX\_MEM\_SIZE

Each remembered trace is a string that is extended as optimization progresses and appends data to it. The optimizer\_trace\_max\_mem\_size variable sets a limit on the total amount of memory used by all currently remembered traces. If this limit is reached, the current trace is not extended (and thus is incomplete), and the MISSING\_BYTES\_BEYOND\_MAX\_MEM\_SIZE column shows the number of bytes missing from the trace.

• INSUFFICIENT\_PRIVILEGES

If a traced query uses views or stored routines that have SQL SECURITY with a value of DEFINER, it may be that a user other than the definer is denied from seeing the trace of the query. In that case, the trace is shown as empty and INSUFFICIENT\_PRIVILEGES has a value of 1. Otherwise, the value is 0.

**26.3.20** **The** **INFORMATION\_SCHEMA** **PARAMETERS** **Table**

The [PARAMETERS](#_bookmark91) table provides information about parameters for stored routines (stored procedures and stored functions), and about return values for stored functions. The [PARAMETERS](#_bookmark91) table does not include built-in (native) functions or loadable functions.

The [PARAMETERS](#_bookmark91) table has these columns:

• SPECIFIC\_CATALOG

The name of the catalog to which the routine containing the parameter belongs. This value is always def.

• SPECIFIC\_SCHEMA

The name of the schema (database) to which the routine containing the parameter belongs.

• SPECIFIC\_NAME

The name of the routine containing the parameter.

• ORDINAL\_POSITION

For successive parameters of a stored procedure or function, the ORDINAL\_POSITION values are 1, 2, 3, and so forth. For a stored function, there is also a row that applies to the function return value (as described by the RETURNS clause). The return value is not a true parameter, so the row that describes it has these unique characteristics:

• The ORDINAL\_POSITION value is 0.

• The PARAMETER\_NAME and PARAMETER\_MODE values are NULL because the return value has no name and the mode does not apply.

• PARAMETER\_MODE

The mode of the parameter. This value is one of IN, OUT, or INOUT. For a stored function return value, this value is NULL.

• PARAMETER\_NAME

The name of the parameter. For a stored function return value, this value is NULL.

• DATA\_TYPE The parameter data type.

The DATA\_TYPE value is the type name only with no other information. The DTD\_IDENTIFIER value contains the type name and possibly other information such as the precision or length.

• CHARACTER\_MAXIMUM\_LENGTH

For string parameters, the maximum length in characters.

• CHARACTER\_OCTET\_LENGTH

For string parameters, the maximum length in bytes.

• NUMERIC\_PRECISION

For numeric parameters, the numeric precision.

• NUMERIC\_SCALE

For numeric parameters, the numeric scale.

• DATETIME\_PRECISION

For temporal parameters, the fractional seconds precision.

• CHARACTER\_SET\_NAME

For character string parameters, the character set name.

• COLLATION\_NAME

For character string parameters, the collation name.

• DTD\_IDENTIFIER The parameter data type.

The DATA\_TYPE value is the type name only with no other information. The DTD\_IDENTIFIER value contains the type name and possibly other information such as the precision or length.

• ROUTINE\_TYPE

PROCEDURE for stored procedures, FUNCTION for stored functions.

**26.3.21** **The** **INFORMATION\_SCHEMA** **PARTITIONS** **Table**

The [PARTITIONS](#_bookmark3) table provides information about table partitions. Each row in this table corresponds to an individual partition or subpartition of a partitioned table. For more information about partitioning tables, see Chapter 24, *Partitioning*.

The [PARTITIONS](#_bookmark3) table has these columns:

• TABLE\_CATALOG

The name of the catalog to which the table belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the table belongs.

• TABLE\_NAME

The name of the table containing the partition.

• PARTITION\_NAME The name of the partition.

• SUBPARTITION\_NAME

If the [PARTITIONS](#_bookmark3) table row represents a subpartition, the name of subpartition; otherwise NULL. For NDB: This value is always NULL.

• PARTITION\_ORDINAL\_POSITION

All partitions are indexed in the same order as they are defined, with 1 being the number assigned to the first partition. The indexing can change as partitions are added, dropped, and reorganized; the number shown is this column reflects the current order, taking into account any indexing changes.

• SUBPARTITION\_ORDINAL\_POSITION

Subpartitions within a given partition are also indexed and reindexed in the same manner as partitions are indexed within a table.

• PARTITION\_METHOD

One of the values RANGE, LIST, HASH, LINEAR HASH, KEY, or LINEAR KEY; that is, one of the available partitioning types as discussed in Section 24.2, “Partitioning Types” .

• SUBPARTITION\_METHOD

One of the values HASH, LINEAR HASH, KEY, or LINEAR KEY; that is, one of the available subpartitioning types as discussed in Section 24.2.6, “Subpartitioning” .

• PARTITION\_EXPRESSION

The expression for the partitioning function used in the CREATE TABLE or ALTER TABLE statement that created the table's current partitioning scheme.

For example, consider a partitioned table created in the test database using this statement:

CREATE TABLE tp (

c1 INT,

c2 INT,

c3 VARCHAR(25)

)

PARTITION BY HASH(c1 + c2)

PARTITIONS 4;

The PARTITION\_EXPRESSION column in a [PARTITIONS](#_bookmark3) table row for a partition from this table displays c1 + c2, as shown here:

mysql> **SELECT** **DISTINCT** **PARTITION\_EXPRESSION**

**FROM** **INFORMATION\_SCHEMA.PARTITIONS**

**WHERE** **TABLE\_NAME='tp'** **AND** **TABLE\_SCHEMA='test';**

+----------------------+

| PARTITION\_EXPRESSION |

+----------------------+

| c1 + c2 |

+----------------------+

For a table that is not explicitly partitioned, this column is always NULL, regardless of storage engine.

• SUBPARTITION\_EXPRESSION

This works in the same fashion for the subpartitioning expression that defines the subpartitioning for a table as PARTITION\_EXPRESSION does for the partitioning expression used to define a table's partitioning.

If the table has no subpartitions, this column is NULL.

• PARTITION\_DESCRIPTION

This column is used for RANGE and LIST partitions. For a RANGE partition, it contains the value set in the partition's VALUES LESS THAN clause, which can be either an integer or MAXVALUE. For a LIST partition, this column contains the values defined in the partition's VALUES IN clause, which is a list of comma-separated integer values.

For partitions whose PARTITION\_METHOD is other than RANGE or LIST, this column is always NULL.

• TABLE\_ROWS

The number of table rows in the partition.

For partitioned InnoDB tables, the row count given in the TABLE\_ROWS column is only an estimated value used in SQL optimization, and may not always be exact.

For NDB tables, you can also obtain this information using the ndb\_desc utility.

• AVG\_ROW\_LENGTH

The average length of the rows stored in this partition or subpartition, in bytes. This is the same as DATA\_LENGTH divided by TABLE\_ROWS.

For NDB tables, you can also obtain this information using the ndb\_desc utility.

• DATA\_LENGTH

The total length of all rows stored in this partition or subpartition, in bytes; that is, the total number of bytes stored in the partition or subpartition.

For NDB tables, you can also obtain this information using the ndb\_desc utility.

• MAX\_DATA\_LENGTH

The maximum number of bytes that can be stored in this partition or subpartition. For NDB tables, you can also obtain this information using the ndb\_desc utility.

• INDEX\_LENGTH

The length of the index file for this partition or subpartition, in bytes.

For partitions of NDB tables, whether the tables use implicit or explicit partitioning, the INDEX\_LENGTH column value is always 0. However, you can obtain equivalent information using the ndb\_desc utility.

• DATA\_FREE

The number of bytes allocated to the partition or subpartition but not used. For NDB tables, you can also obtain this information using the ndb\_desc utility.

• CREATE\_TIME

The time that the partition or subpartition was created.

• UPDATE\_TIME

The time that the partition or subpartition was last modified.

• CHECK\_TIME

The last time that the table to which this partition or subpartition belongs was checked. For partitioned InnoDB tables, the value is always NULL.

• CHECKSUM

The checksum value, if any; otherwise NULL.

• PARTITION\_COMMENT

The text of the comment, if the partition has one. If not, this value is empty.

The maximum length for a partition comment is defined as 1024 characters, and the display width of the PARTITION\_COMMENT column is also 1024, characters to match this limit.

• NODEGROUP

This is the nodegroup to which the partition belongs. For NDB Cluster tables, this is always default. For partitioned tables using storage engines other than NDB, the value is also default. Otherwise, this column is empty.

• TABLESPACE\_NAME

The name of the tablespace to which the partition belongs. The value is always DEFAULT, unless the table uses the NDB storage engine (see the *Notes* at the end of this section).

**Notes**

• [PARTITIONS](#_bookmark3) is a nonstandard INFORMATION\_SCHEMA table.

• A table using any storage engine other than NDB and which is not partitioned has one row in the [PARTITIONS](#_bookmark3) table. However, the values of the PARTITION\_NAME, SUBPARTITION\_NAME, PARTITION\_ORDINAL\_POSITION, SUBPARTITION\_ORDINAL\_POSITION, PARTITION\_METHOD, SUBPARTITION\_METHOD, PARTITION\_EXPRESSION, SUBPARTITION\_EXPRESSION, and PARTITION\_DESCRIPTION columns are all NULL. Also, the PARTITION\_COMMENT column in this case is blank.

• An NDB table which is not explicitly partitioned has one row in the PARTITIONS table for each data node in the NDB cluster. For each such row:

• The SUBPARTITION\_NAME, SUBPARTITION\_ORDINAL\_POSITION, SUBPARTITION\_METHOD, PARTITION\_EXPRESSION, SUBPARTITION\_EXPRESSION, CREATE\_TIME, UPDATE\_TIME, CHECK\_TIME, CHECKSUM, and TABLESPACE\_NAME columns are all NULL.

• The PARTITION\_METHOD is always AUTO.

• The NODEGROUP column is default.

• The PARTITION\_COMMENT column is empty.

**26.3.22** **The** **INFORMATION\_SCHEMA** **PLUGINS** **Table**

The [PLUGINS](#_bookmark92) table provides information about server plugins.

The [PLUGINS](#_bookmark92) table has these columns:

• PLUGIN\_NAME

The name used to refer to the plugin in statements such as INSTALL PLUGIN and UNINSTALL

PLUGIN.

• PLUGIN\_VERSION

The version from the plugin's general type descriptor.

• PLUGIN\_STATUS

The plugin status, one of ACTIVE, INACTIVE, DISABLED, DELETING, or DELETED.

• PLUGIN\_TYPE

The type of plugin, such as STORAGE ENGINE, INFORMATION\_SCHEMA, or AUTHENTICATION.

• PLUGIN\_TYPE\_VERSION

The version from the plugin's type-specific descriptor.

• PLUGIN\_LIBRARY

The name of the plugin shared library file. This is the name used to refer to the plugin file in statements such as INSTALL PLUGIN and UNINSTALL PLUGIN. This file is located in the directory named by the plugin\_dir system variable. If the library name is NULL, the plugin is compiled in and cannot be uninstalled with UNINSTALL PLUGIN.

• PLUGIN\_LIBRARY\_VERSION The plugin API interface version.

• PLUGIN\_AUTHOR The plugin author.

• PLUGIN\_DESCRIPTION

A short description of the plugin.

• PLUGIN\_LICENSE

How the plugin is licensed (for example, GPL).

• LOAD\_OPTION

How the plugin was loaded. The value is OFF, ON, FORCE, or FORCE\_PLUS\_PERMANENT. See Section 5.6.1, “Installing and Uninstalling Plugins” .

**Notes**

• [PLUGINS](#_bookmark92) is a nonstandard INFORMATION\_SCHEMA table.

• For plugins installed with INSTALL PLUGIN, the PLUGIN\_NAME and PLUGIN\_LIBRARY values are also registered in the mysql.plugin table.

• For information about plugin data structures that form the basis of the information in the [PLUGINS](#_bookmark92) table, see [The MySQL Plugin API](https://dev.mysql.com/doc/extending-mysql/8.0/en/plugin-api.html).

Plugin information is also available from the SHOW PLUGINS statement. See Section 13.7.7.25, “SHOW PLUGINS Statement” . These statements are equivalent:

SELECT



PLUGIN\_NAME, PLUGIN\_STATUS, PLUGIN\_TYPE,

PLUGIN\_LIBRARY, PLUGIN\_LICENSE

FROM INFORMATION\_SCHEMA.PLUGINS;

SHOW PLUGINS;

**26.3.23** **The** **INFORMATION\_SCHEMA** **PROCESSLIST** **Table**

The MySQL process list indicates the operations currently being performed by the set of threads executing within the server. The [PROCESSLIST](#_bookmark93) table is one source of process information. For a

comparison of this table with other sources, see Sources of Process Information.

The [PROCESSLIST](#_bookmark93) table has these columns:

• ID

The connection identifier. This is the same value displayed in the Id column of the SHOW PROCESSLIST statement, displayed in the PROCESSLIST\_ID column of the Performance Schema threads table, and returned by the CONNECTION\_ID() function within the thread.

• USER

The MySQL user who issued the statement. A value of system user refers to a nonclient thread spawned by the server to handle tasks internally, for example, a delayed-row handler thread or an I/O or SQL thread used on replica hosts. For system user, there is no host specified in the Host column. unauthenticated user refers to a thread that has become associated with a client connection but for which authentication of the client user has not yet occurred. event\_scheduler refers to the thread that monitors scheduled events (see [Section 25.4, “Using the Event Scheduler”](#_bookmark33)).

**Note**

A USER value of system user is distinct from the SYSTEM\_USER privilege. The former designates internal threads. The latter distinguishes the system user and regular user account categories (see Section 6.2.11, “Account Categories”).

• HOST

The host name of the client issuing the statement (except for system user, for which there is no host). The host name for TCP/IP connections is reported in *host\_name*:*client\_port* format to make it easier to determine which client is doing what.

• DB

The default database for the thread, or NULL if none has been selected.

• COMMAND

The type of command the thread is executing on behalf of the client, or Sleep if the session is idle. For descriptions of thread commands, see Section 8.14, “Examining Server Thread (Process) Information” . The value of this column corresponds to the COM\_*xxx* commands of the client/server protocol and Com\_*xxx* status variables. See Section 5.1.10, “Server Status Variables” .

• TIME

The time in seconds that the thread has been in its current state. For a replica SQL thread, the value is the number of seconds between the timestamp of the last replicated event and the real time of the replica host. See Section 17.2.3, “Replication Threads” .

• STATE

An action, event, or state that indicates what the thread is doing. For descriptions of STATE values, see Section 8.14, “Examining Server Thread (Process) Information” .



Most states correspond to very quick operations. If a thread stays in a given state for many seconds, there might be a problem that needs to be investigated.

• INFO

The statement the thread is executing, or NULL if it is executing no statement. The statement might be the one sent to the server, or an innermost statement if the statement executes other statements. For example, if a CALL statement executes a stored procedure that is executing a SELECT statement, the INFO value shows the SELECT statement.

**Notes**

• [PROCESSLIST](#_bookmark93) is a nonstandard INFORMATION\_SCHEMA table.

• Like the output from the SHOW PROCESSLIST statement, the [PROCESSLIST](#_bookmark93) table provides information about all threads, even those belonging to other users, if you have the PROCESS privilege. Otherwise (without the PROCESS privilege), nonanonymous users have access to information about their own threads but not threads for other users, and anonymous users have no access to thread information.

• If an SQL statement refers to the [PROCESSLIST](#_bookmark93) table, MySQL populates the entire table once, when statement execution begins, so there is read consistency during the statement. There is no read consistency for a multi-statement transaction.

The following statements are equivalent:

SELECT \* FROM INFORMATION\_SCHEMA.PROCESSLIST

SHOW FULL PROCESSLIST

**26.3.24** **The** **INFORMATION\_SCHEMA** **PROFILING** **Table**

The [PROFILING](#_bookmark94) table provides statement profiling information. Its contents correspond to the information produced by the SHOW PROFILE and SHOW PROFILES statements (see Section 13.7.7.30, “SHOW PROFILE Statement” ). The table is empty unless the profiling session variable is set to 1.

**Note**

This table is deprecated; expect it to be removed in a future MySQL release. Use the [Performance Schema](#_bookmark171) instead; see Section 27.19.1, “Query Profiling Using Performance Schema” .

The [PROFILING](#_bookmark94) table has these columns:

• QUERY\_ID

A numeric statement identifier.

• SEQ

A sequence number indicating the display order for rows with the same QUERY\_ID value.

• STATE

The profiling state to which the row measurements apply.

• DURATION

How long statement execution remained in the given state, in seconds.

• CPU\_USER, CPU\_SYSTEM

User and system CPU use, in seconds.

• CONTEXT\_VOLUNTARY, CONTEXT\_INVOLUNTARY

How many voluntary and involuntary context switches occurred.

• BLOCK\_OPS\_IN, BLOCK\_OPS\_OUT

The number of block input and output operations.

• MESSAGES\_SENT, MESSAGES\_RECEIVED

The number of communication messages sent and received.

• PAGE\_FAULTS\_MAJOR, PAGE\_FAULTS\_MINOR The number of major and minor page faults.

• SWAPS

How many swaps occurred.

• SOURCE\_FUNCTION, SOURCE\_FILE, and SOURCE\_LINE

Information indicating where in the source code the profiled state executes.

**Notes**

• [PROFILING](#_bookmark94) is a nonstandard INFORMATION\_SCHEMA table.

Profiling information is also available from the SHOW PROFILE and SHOW PROFILES statements. See Section 13.7.7.30, “SHOW PROFILE Statement” . For example, the following queries are equivalent:

SHOW PROFILE FOR QUERY 2;

SELECT STATE, FORMAT(DURATION, 6) AS DURATION

FROM INFORMATION\_SCHEMA.PROFILING

WHERE QUERY\_ID = 2 ORDER BY SEQ;

**26.3.25** **The** **INFORMATION\_SCHEMA** **REFERENTIAL\_CONSTRAINTS** **Table**

The [REFERENTIAL\_CONSTRAINTS](#_bookmark95) table provides information about foreign keys. The [REFERENTIAL\_CONSTRAINTS](#_bookmark95) table has these columns:

• CONSTRAINT\_CATALOG

The name of the catalog to which the constraint belongs. This value is always def.

• CONSTRAINT\_SCHEMA

The name of the schema (database) to which the constraint belongs.

• CONSTRAINT\_NAME The name of the constraint.

• UNIQUE\_CONSTRAINT\_CATALOG

The name of the catalog containing the unique constraint that the constraint references. This value is always def.

• UNIQUE\_CONSTRAINT\_SCHEMA

The name of the schema containing the unique constraint that the constraint references.

• UNIQUE\_CONSTRAINT\_NAME

The name of the unique constraint that the constraint references.

• MATCH\_OPTION

The value of the constraint MATCH attribute. The only valid value at this time is NONE.

• UPDATE\_RULE

The value of the constraint ON UPDATE attribute. The possible values are CASCADE, SET NULL, SET

DEFAULT, RESTRICT, NO ACTION.

• DELETE\_RULE

The value of the constraint ON DELETE attribute. The possible values are CASCADE, SET NULL, SET

DEFAULT, RESTRICT, NO ACTION.

• TABLE\_NAME

The name of the table. This value is the same as in the [TABLE\_CONSTRAINTS](#_bookmark111) table.

• REFERENCED\_TABLE\_NAME

The name of the table referenced by the constraint.

**26.3.26** **The** **INFORMATION\_SCHEMA** **RESOURCE\_GROUPS** **Table**

The [RESOURCE\_GROUPS](#_bookmark96) table provides access to information about resource groups. For general discussion of the resource group capability, see Section 5.1.16, “Resource Groups” .

You can see information only for columns for which you have some privilege.

The [RESOURCE\_GROUPS](#_bookmark96) table has these columns:

• RESOURCE\_GROUP\_NAME The name of the resource group.

• RESOURCE\_GROUP\_TYPE

The resource group type, either SYSTEM or USER.

• RESOURCE\_GROUP\_ENABLED

Whether the resource group is enabled (1) or disabled (0);

• VCPU\_IDS

The CPU affinity; that is, the set of virtual CPUs that the resource group can use. The value is a list of comma-separated CPU numbers or ranges.

• THREAD\_PRIORITY

The priority for threads assigned to the resource group. The priority ranges from -20 (highest priority) to 19 (lowest priority). System resource groups have a priority that ranges from -20 to 0. User resource groups have a priority that ranges from 0 to 19.

**26.3.27** **The** **INFORMATION\_SCHEMA** **ROLE\_COLUMN\_GRANTS** **Table**

The [ROLE\_COLUMN\_GRANTS](#_bookmark97) table (available as of MySQL 8.0.19) provides information about the column privileges for roles that are available to or granted by the currently enabled roles.

The [ROLE\_COLUMN\_GRANTS](#_bookmark97) table has these columns:

• GRANTOR

The user name part of the account that granted the role.

• GRANTOR\_HOST

The host name part of the account that granted the role.

• GRANTEE

The user name part of the account to which the role is granted.

• GRANTEE\_HOST

The host name part of the account to which the role is granted.

• TABLE\_CATALOG

The name of the catalog to which the role applies. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the role applies.

• TABLE\_NAME

The name of the table to which the role applies.

• COLUMN\_NAME

The name of the column to which the role applies.

• PRIVILEGE\_TYPE

The privilege granted. The value can be any privilege that can be granted at the column level; see Section 13.7.1.6, “GRANT Statement” . Each row lists a single privilege, so there is one row per column privilege held by the grantee.

• IS\_GRANTABLE

YES or NO, depending on whether the role is grantable to other accounts.

**26.3.28** **The** **INFORMATION\_SCHEMA** **ROLE\_ROUTINE\_GRANTS** **Table**

The [ROLE\_ROUTINE\_GRANTS](#_bookmark98) table (available as of MySQL 8.0.19) provides information about the routine privileges for roles that are available to or granted by the currently enabled roles.

The [ROLE\_ROUTINE\_GRANTS](#_bookmark98) table has these columns:

• GRANTOR

The user name part of the account that granted the role.

• GRANTOR\_HOST

The host name part of the account that granted the role.

• GRANTEE

The user name part of the account to which the role is granted.

• GRANTEE\_HOST

The host name part of the account to which the role is granted.

• SPECIFIC\_CATALOG

The name of the catalog to which the routine belongs. This value is always def.

• SPECIFIC\_SCHEMA

The name of the schema (database) to which the routine belongs.

• SPECIFIC\_NAME The name of the routine.

• ROUTINE\_CATALOG

The name of the catalog to which the routine belongs. This value is always def.

• ROUTINE\_SCHEMA

The name of the schema (database) to which the routine belongs.

• ROUTINE\_NAME The name of the routine.

• PRIVILEGE\_TYPE

The privilege granted. The value can be any privilege that can be granted at the routine level; see Section 13.7.1.6, “GRANT Statement” . Each row lists a single privilege, so there is one row per column privilege held by the grantee.

• IS\_GRANTABLE

YES or NO, depending on whether the role is grantable to other accounts.

**26.3.29** **The** **INFORMATION\_SCHEMA** **ROLE\_TABLE\_GRANTS** **Table**

The [ROLE\_TABLE\_GRANTS](#_bookmark99) table (available as of MySQL 8.0.19) provides information about the table privileges for roles that are available to or granted by the currently enabled roles.

The [ROLE\_TABLE\_GRANTS](#_bookmark99) table has these columns:

• GRANTOR

The user name part of the account that granted the role.

• GRANTOR\_HOST

The host name part of the account that granted the role.

• GRANTEE

The user name part of the account to which the role is granted.

• GRANTEE\_HOST

The host name part of the account to which the role is granted.

• TABLE\_CATALOG

The name of the catalog to which the role applies. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the role applies.

• TABLE\_NAME

The name of the table to which the role applies.

• PRIVILEGE\_TYPE

The privilege granted. The value can be any privilege that can be granted at the table level; see Section 13.7.1.6, “GRANT Statement” . Each row lists a single privilege, so there is one row per column privilege held by the grantee.

• IS\_GRANTABLE

YES or NO, depending on whether the role is grantable to other accounts.

**26.3.30** **The** **INFORMATION\_SCHEMA** **ROUTINES** **Table**

The [ROUTINES](#_bookmark50) table provides information about stored routines (stored procedures and stored functions). The [ROUTINES](#_bookmark50) table does not include built-in (native) functions or loadable functions.

The [ROUTINES](#_bookmark50) table has these columns:

• SPECIFIC\_NAME The name of the routine.

• ROUTINE\_CATALOG

The name of the catalog to which the routine belongs. This value is always def.

• ROUTINE\_SCHEMA

The name of the schema (database) to which the routine belongs.

• ROUTINE\_NAME The name of the routine.

• ROUTINE\_TYPE

PROCEDURE for stored procedures, FUNCTION for stored functions.

• DATA\_TYPE

If the routine is a stored function, the return value data type. If the routine is a stored procedure, this value is empty.

The DATA\_TYPE value is the type name only with no other information. The DTD\_IDENTIFIER value contains the type name and possibly other information such as the precision or length.

• CHARACTER\_MAXIMUM\_LENGTH

For stored function string return values, the maximum length in characters. If the routine is a stored procedure, this value is NULL.

• CHARACTER\_OCTET\_LENGTH

For stored function string return values, the maximum length in bytes. If the routine is a stored procedure, this value is NULL.

• NUMERIC\_PRECISION

For stored function numeric return values, the numeric precision. If the routine is a stored procedure, this value is NULL.

• NUMERIC\_SCALE

For stored function numeric return values, the numeric scale. If the routine is a stored procedure, this value is NULL.

• DATETIME\_PRECISION

For stored function temporal return values, the fractional seconds precision. If the routine is a stored procedure, this value is NULL.

• CHARACTER\_SET\_NAME

For stored function character string return values, the character set name. If the routine is a stored procedure, this value is NULL.

• COLLATION\_NAME

For stored function character string return values, the collation name. If the routine is a stored procedure, this value is NULL.

• DTD\_IDENTIFIER

If the routine is a stored function, the return value data type. If the routine is a stored procedure, this value is empty.

The DATA\_TYPE value is the type name only with no other information. The DTD\_IDENTIFIER value contains the type name and possibly other information such as the precision or length.

• ROUTINE\_BODY

The language used for the routine definition. This value is always SQL.

• ROUTINE\_DEFINITION

The text of the SQL statement executed by the routine.

• EXTERNAL\_NAME

This value is always NULL.

• EXTERNAL\_LANGUAGE

The language of the stored routine. The value is read from the external\_language column of the mysql.routines data dictionary table.

• PARAMETER\_STYLE

This value is always SQL.

• IS\_DETERMINISTIC

YES or NO, depending on whether the routine is defined with the DETERMINISTIC characteristic.

• SQL\_DATA\_ACCESS

The data access characteristic for the routine. The value is one of CONTAINS SQL, NO SQL, READS SQL DATA, or MODIFIES SQL DATA.

• SQL\_PATH

This value is always NULL.

• SECURITY\_TYPE

The routine SQL SECURITY characteristic. The value is one of DEFINER or INVOKER.

• CREATED

The date and time when the routine was created. This is a TIMESTAMP value.

• LAST\_ALTERED

The date and time when the routine was last modified. This is a TIMESTAMP value. If the routine has not been modified since its creation, this value is the same as the CREATED value.

• SQL\_MODE

The SQL mode in effect when the routine was created or altered, and under which the routine executes. For the permitted values, see Section 5.1.11, “Server SQL Modes” .

• ROUTINE\_COMMENT

The text of the comment, if the routine has one. If not, this value is empty.

• DEFINER

The account named in the DEFINER clause (often the user who created the routine), in '*user\_name*'@'*host\_name*' format.

• CHARACTER\_SET\_CLIENT

The session value of the character\_set\_client system variable when the routine was created.

• COLLATION\_CONNECTION

The session value of the collation\_connection system variable when the routine was created.

• DATABASE\_COLLATION

The collation of the database with which the routine is associated.

**Notes**

• To see information about a routine, you must be the user named as the routine DEFINER, have the SHOW\_ROUTINE privilege, have the SELECT privilege at the global level, or have the CREATE ROUTINE, ALTER ROUTINE, or EXECUTE privilege granted at a scope that includes the routine. The ROUTINE\_DEFINITION column is NULL if you have only CREATE ROUTINE, ALTER ROUTINE, or EXECUTE.

• Information about stored function return values is also available in the [PARAMETERS](#_bookmark91) table. The return value row for a stored function can be identified as the row that has an ORDINAL\_POSITION value of 0.

**26.3.31** **The** **INFORMATION\_SCHEMA** **SCHEMATA** **Table**

A schema is a database, so the [SCHEMATA](#_bookmark100) table provides information about databases. The [SCHEMATA](#_bookmark100) table has these columns:

• CATALOG\_NAME

The name of the catalog to which the schema belongs. This value is always def.

• SCHEMA\_NAME

The name of the schema.

• DEFAULT\_CHARACTER\_SET\_NAME The schema default character set.

• DEFAULT\_COLLATION\_NAME The schema default collation.

• SQL\_PATH

This value is always NULL.

• DEFAULT\_ENCRYPTION

The schema default encryption. This column was added in MySQL 8.0.16.

Schema names are also available from the SHOW DATABASES statement. See Section 13.7.7.14, “SHOW DATABASES Statement” . The following statements are equivalent:

SELECT SCHEMA\_NAME AS `Database`

FROM INFORMATION\_SCHEMA.SCHEMATA

[WHERE SCHEMA\_NAME LIKE '*wild*']

SHOW DATABASES

[LIKE '*wild*']

You see only those databases for which you have some kind of privilege, unless you have the global SHOW DATABASES privilege.



**Notes**



**Caution**

Because any static global privilege is considered a privilege for all databases, any static global privilege enables a user to see all database names with SHOW DATABASES or by examining the [SCHEMATA](#_bookmark100) table of INFORMATION\_SCHEMA, except databases that have been restricted at the database level by partial revokes.

• The [SCHEMATA EXTENSIONS](#_bookmark101)\_ table augments the [SCHEMATA](#_bookmark100) table with information about schema options.

**26.3.32** **The** **INFORMATION\_SCHEMA** **SCHEMATA\_EXTENSIONS** **Table**

The [SCHEMATA EXTENSIONS](#_bookmark101)\_ table (available as of MySQL 8.0.22) augments the [SCHEMATA](#_bookmark100) table with information about schema options.

The [SCHEMATA\_EXTENSIONS](#_bookmark101) table has these columns:

• CATALOG\_NAME

The name of the catalog to which the schema belongs. This value is always def.

• SCHEMA\_NAME

The name of the schema.

• OPTIONS

The options for the schema. If the schema is read only, the value contains READ ONLY=1. If the schema is not read only, no READ ONLY option appears.

**Example**

mysql> **ALTER** **SCHEMA** **mydb** **READ** **ONLY** **=** **1;**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.SCHEMATA\_EXTENSIONS**

**WHERE** **SCHEMA\_NAME** **=** **'mydb';**

+--------------+-------------+-------------+

| CATALOG\_NAME | SCHEMA\_NAME | OPTIONS |

**WHERE** **SCHEMA\_NAME** **=** **'mydb';**

+--------------+-------------+---------+

| CATALOG\_NAME | SCHEMA\_NAME | OPTIONS |

+--------------+-------------+---------+

| def | mydb | |

+--------------+-------------+---------+

+--------------+-------------+-------------+

| def | mydb | READ ONLY=1 |

+--------------+-------------+-------------+

mysql> **ALTER** **SCHEMA** **mydb** **READ** **ONLY** **=** **0;**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.SCHEMATA\_EXTENSIONS**

**Notes**

• [SCHEMATA\_EXTENSIONS](#_bookmark101) is a nonstandard INFORMATION\_SCHEMA table.

**26.3.33** **The** **INFORMATION\_SCHEMA** **SCHEMA\_PRIVILEGES** **Table**

The [SCHEMA\_PRIVILEGES](#_bookmark102) table provides information about schema (database) privileges. It takes its

values from the mysql.db system table.

The [SCHEMA\_PRIVILEGES](#_bookmark102) table has these columns:

• GRANTEE

The name of the account to which the privilege is granted, in '*user\_name*'@'*host\_name*' format.

• TABLE\_CATALOG

The name of the catalog to which the schema belongs. This value is always def.

• TABLE\_SCHEMA The name of the schema.

• PRIVILEGE\_TYPE

The privilege granted. The value can be any privilege that can be granted at the schema level; see Section 13.7.1.6, “GRANT Statement” . Each row lists a single privilege, so there is one row per schema privilege held by the grantee.

• IS\_GRANTABLE

YES if the user has the GRANT OPTION privilege, NO otherwise. The output does not list GRANT OPTION as a separate row with PRIVILEGE\_TYPE='GRANT OPTION'.

**Notes**

• [SCHEMA\_PRIVILEGES](#_bookmark102) is a nonstandard INFORMATION\_SCHEMA table.

The following statements are *not* equivalent:

SELECT ... FROM INFORMATION\_SCHEMA.SCHEMA\_PRIVILEGES

SHOW GRANTS ...

**26.3.34** **The** **INFORMATION\_SCHEMA** **STATISTICS** **Table**

The [STATISTICS](#_bookmark103) table provides information about table indexes.

Columns in [STATISTICS](#_bookmark103) that represent table statistics hold cached values. The information\_schema\_stats\_expiry system variable defines the period of time before cached table statistics expire. The default is 86400 seconds (24 hours). If there are no cached statistics or statistics have expired, statistics are retrieved from storage engines when



querying table statistics columns. To update cached values at any time for a given table, use ANALYZE TABLE. To always retrieve the latest statistics directly from storage engines, set information\_schema\_stats\_expiry=0. For more information, see Section 8.2.3, “Optimizing INFORMATION\_SCHEMA Queries” .

**Note**

If the innodb\_read\_only system variable is enabled, ANALYZE TABLE may fail because it cannot update statistics tables in the data dictionary,

which use InnoDB. For ANALYZE TABLE operations that update the key distribution, failure may occur even if the operation updates the table itself (for example, if it is a MyISAM table). To obtain the updated distribution statistics, set information\_schema\_stats\_expiry=0.

The [STATISTICS](#_bookmark103) table has these columns:

• TABLE\_CATALOG

The name of the catalog to which the table containing the index belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the table containing the index belongs.

• TABLE\_NAME

The name of the table containing the index.

• NON\_UNIQUE

0 if the index cannot contain duplicates, 1 if it can.

• INDEX\_SCHEMA

The name of the schema (database) to which the index belongs.

• INDEX\_NAME

The name of the index. If the index is the primary key, the name is always PRIMARY.

• SEQ\_IN\_INDEX

The column sequence number in the index, starting with 1.

• COLUMN\_NAME

The column name. See also the description for the EXPRESSION column.

• COLLATION

How the column is sorted in the index. This can have values A (ascending), D (descending), or NULL (not sorted).

• CARDINALITY

An estimate of the number of unique values in the index. To update this number, run ANALYZE TABLE or (for MyISAM tables) myisamchk -a.

CARDINALITY is counted based on statistics stored as integers, so the value is not necessarily exact even for small tables. The higher the cardinality, the greater the chance that MySQL uses the index when doing joins.

• SUB\_PART



The index prefix. That is, the number of indexed characters if the column is only partly indexed, NULL if the entire column is indexed.

**Note**

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.

For additional information about index prefixes, see Section 8.3.5, “Column Indexes” , and Section 13.1.15, “CREATE INDEX Statement” .

• PACKED

Indicates how the key is packed. NULL if it is not.

• NULLABLE

Contains YES if the column may contain NULL values and '' if not.

• INDEX\_TYPE

The index method used (BTREE, FULLTEXT, HASH, RTREE).

• COMMENT

Information about the index not described in its own column, such as disabled if the index is disabled.

• INDEX\_COMMENT

Any comment provided for the index with a COMMENT attribute when the index was created.

• IS\_VISIBLE

Whether the index is visible to the optimizer. See Section 8.3.12, “Invisible Indexes” .

• EXPRESSION

MySQL 8.0.13 and higher supports functional key parts (see Functional Key Parts), which affects both the COLUMN\_NAME and EXPRESSION columns:

• For a nonfunctional key part, COLUMN\_NAME indicates the column indexed by the key part and EXPRESSION is NULL.

• For a functional key part, COLUMN\_NAME column is NULL and EXPRESSION indicates the expression for the key part.

**Notes**

• There is no standard INFORMATION\_SCHEMA table for indexes. The MySQL column list is similar to what SQL Server 2000 returns for sp\_statistics, except that QUALIFIER and OWNER are replaced with CATALOG and SCHEMA, respectively.

Information about table indexes is also available from the SHOW INDEX statement. See Section 13.7.7.22, “SHOW INDEX Statement” . The following statements are equivalent:

SELECT \* FROM INFORMATION\_SCHEMA.STATISTICS

WHERE table\_name = '*tbl\_name* '

AND table\_schema = '*db\_name* '

SHOW INDEX

FROM *tbl\_name*

FROM *db\_name*

In MySQL 8.0.30 and later, information about generated invisible primary key columns

is visible in this table by default. You can cause such information to be hidden by setting show\_gipk\_in\_create\_table\_and\_information\_schema = OFF. For more information, see Section 13. 1.20. 11, “Generated Invisible Primary Keys” .

**26.3.35** **The** **INFORMATION\_SCHEMA** **ST\_GEOMETRY\_COLUMNS** **Table**

The [ST\_GEOMETRY\_COLUMNS](#_bookmark104) table provides information about table columns that store spatial data. This table is based on the SQL/MM (ISO/IEC 13249-3) standard, with extensions as noted. MySQL implements [ST GEOMETRY COLUMNS](#_bookmark104)\_\_ as a view on the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table.

The [ST\_GEOMETRY\_COLUMNS](#_bookmark104) table has these columns:

• TABLE\_CATALOG

The name of the catalog to which the table containing the column belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the table containing the column belongs.

• TABLE\_NAME

The name of the table containing the column.

• COLUMN\_NAME The name of the column.

• SRS\_NAME

The spatial reference system (SRS) name.

• SRS\_ID

The spatial reference system ID (SRID).

• GEOMETRY\_TYPE\_NAME

The column data type. Permitted values are: geometry, point, linestring, polygon, multipoint, multilinestring, multipolygon, geometrycollection. This column is a MySQL extension to the standard.

**26.3.36** **The** **INFORMATION\_SCHEMA** **ST\_SPATIAL\_REFERENCE\_SYSTEMS**

**Table**

The [ST\_SPATIAL\_REFERENCE\_SYSTEMS](#_bookmark105) table provides information about available spatial reference systems (SRSs) for spatial data. This table is based on the SQL/MM (ISO/IEC 13249-3) standard.

Entries in the [ST\_SPATIAL\_REFERENCE\_SYSTEMS](#_bookmark105) table are based on the [European Petroleum](http://epsg.org) [Survey Group](http://epsg.org) (EPSG) data set, except for SRID 0, which corresponds to a special SRS used in MySQL that represents an infinite flat Cartesian plane with no units assigned to its axes. For additional information about SRSs, see Section 11.4.5, “Spatial Reference System Support” .

The [ST\_SPATIAL\_REFERENCE\_SYSTEMS](#_bookmark105) table has these columns:

• SRS\_NAME

The spatial reference system name. This value is unique.

• SRS\_ID

The spatial reference system numeric ID. This value is unique.

SRS\_ID values represent the same kind of values as the SRID of geometry values or passed as the SRID argument to spatial functions. SRID 0 (the unitless Cartesian plane) is special. It is always a legal spatial reference system ID and can be used in any computations on spatial data that depend on SRID values.

• ORGANIZATION

The name of the organization that defined the coordinate system on which the spatial reference system is based.

• ORGANIZATION\_COORDSYS\_ID

The numeric ID given to the spatial reference system by the organization that defined it.

• DEFINITION

The spatial reference system definition. DEFINITION values are WKT values, represented as specified in the [Open Geospatial Consortium](http://www.opengeospatial.org) document [OGC 12-063r5](http://docs.opengeospatial.org/is/12-063r5/12-063r5.html).

SRS definition parsing occurs on demand when definitions are needed by GIS functions. Parsed definitions are stored in the data dictionary cache to enable reuse and avoid incurring parsing overhead for every statement that needs SRS information.

• DESCRIPTION

The spatial reference system description.

**Notes**

• The SRS\_NAME, ORGANIZATION, ORGANIZATION\_COORDSYS\_ID, and DESCRIPTION columns contain information that may be of interest to users, but they are not used by MySQL.

**Example**

mysql> **SELECT** **\*** **FROM** **ST\_SPATIAL\_REFERENCE\_SYSTEMS**

**WHERE** **SRS\_ID** **=** **4326\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SRS\_NAME: WGS 84

SRS\_ID: 4326

ORGANIZATION: EPSG

ORGANIZATION\_COORDSYS\_ID: 4326

DEFINITION: GEOGCS["WGS 84",DATUM["World Geodetic System 1984",

SPHEROID["WGS 84",6378137,298.257223563,

AUTHORITY["EPSG","7030"]],AUTHORITY["EPSG","6326"]],

PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],

UNIT["degree",0.017453292519943278,

AUTHORITY["EPSG","9122"]],

AXIS["Lat",NORTH],AXIS["Long",EAST],

AUTHORITY["EPSG","4326"]]

DESCRIPTION:

This entry describes the SRS used for GPS systems. It has a name (SRS\_NAME) of WGS 84 and an ID (SRS\_ID) of 4326, which is the ID used by the [European Petroleum Survey Group](http://epsg.org) (EPSG).

The DEFINITION values for projected and geographic SRSs begin with PROJCS and GEOGCS, respectively. The definition for SRID 0 is special and has an empty DEFINITION value. The following query determines how many entries in the [ST\_SPATIAL\_REFERENCE\_SYSTEMS](#_bookmark105) table correspond to projected, geographic, and other SRSs, based on DEFINITION values:

mysql> **SELECT**

**COUNT(\*),**

**CASE** **LEFT(DEFINITION,** **6)**



**WHEN** **'PROJCS'** **THEN** **'Projected'**

**WHEN** **'GEOGCS'** **THEN** **'Geographic'**

**ELSE** **'Other'**

**END** **AS** **SRS\_TYPE**

**FROM** **INFORMATION\_SCHEMA.ST\_SPATIAL\_REFERENCE\_SYSTEMS**

**GROUP** **BY** **SRS\_TYPE;**

+----------+------------+

| COUNT(\*) | SRS\_TYPE |

+----------+------------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | 1  4668  483 | |  |  | | Other  Projected  Geographic | |  |  | |

+----------+------------+

To enable manipulation of SRS entries stored in the data dictionary, MySQL provides these SQL statements:

• CREATE SPATIAL REFERENCE SYSTEM: See Section 13.1.19, “CREATE SPATIAL REFERENCE SYSTEM Statement” . The description for this statement includes additional information about SRS components.

• DROP SPATIAL REFERENCE SYSTEM: See Section 13.1.31, “DROP SPATIAL REFERENCE SYSTEM Statement” .

**26.3.37** **The** **INFORMATION\_SCHEMA** **ST\_UNITS\_OF\_MEASURE** **Table**

The [ST\_UNITS\_OF\_MEASURE](#_bookmark106) table (available as of MySQL 8.0.14) provides information about acceptable units for the ST\_Distance() function.

The [ST\_UNITS\_OF\_MEASURE](#_bookmark106) table has these columns:

• UNIT\_NAME The name of the unit.

• UNIT\_TYPE

The unit type (for example, LINEAR).

• CONVERSION\_FACTOR

A conversion factor used for internal calculations.

• DESCRIPTION

A description of the unit.

**26.3.38** **The** **INFORMATION\_SCHEMA** **TABLES** **Table**

The [TABLES](#_bookmark107) table provides information about tables in databases.

Columns in [TABLES](#_bookmark107) that represent table statistics hold cached values. The information\_schema\_stats\_expiry system variable defines the period of time before cached table statistics expire. The default is 86400 seconds (24 hours). If there are no cached statistics or statistics have expired, statistics are retrieved from storage engines when querying table statistics columns. To update cached values at any time for a given table, use ANALYZE TABLE. To always retrieve the latest statistics directly from storage engines, set information\_schema\_stats\_expiry to 0. For more information, see Section 8.2.3, “Optimizing INFORMATION\_SCHEMA Queries” .

**Note**

If the innodb\_read\_only system variable is enabled, ANALYZE TABLE may fail because it cannot update statistics tables in the data dictionary, which use InnoDB. For ANALYZE TABLE operations that update the key distribution, failure may occur even if the operation updates the table itself (for

 example, if it is a MyISAM table). To obtain the updated distribution statistics, set

information\_schema\_stats\_expiry=0. The [TABLES](#_bookmark107) table has these columns:

• TABLE\_CATALOG

The name of the catalog to which the table belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the table belongs.

• TABLE\_NAME The name of the table.

• TABLE\_TYPE

BASE TABLE for a table, VIEW for a view, or SYSTEM VIEW for an INFORMATION\_SCHEMA table. The [TABLES](#_bookmark107) table does not list TEMPORARY tables.

• ENGINE

The storage engine for the table. See Chapter 15, *The* *InnoDB* *Storage* *Engine*, and Chapter 16, *Alternative* *Storage* *Engines*.

For partitioned tables, ENGINE shows the name of the storage engine used by all partitions.

• VERSION

This column is unused. With the removal of .frm files in MySQL 8.0, this column now reports a hardcoded value of 10, which is the last .frm file version used in MySQL 5.7.

• ROW\_FORMAT

The row-storage format (Fixed, Dynamic, Compressed, Redundant, Compact). For MyISAM tables, Dynamic corresponds to what myisamchk -dvv reports as Packed.

• TABLE\_ROWS

The number of rows. Some storage engines, such as MyISAM, store the exact count. For other storage engines, such as InnoDB, this value is an approximation, and may vary from the actual value by as much as 40% to 50%. In such cases, use SELECT COUNT(\*) to obtain an accurate count.

TABLE\_ROWS is NULL for INFORMATION\_SCHEMA tables.

For InnoDB tables, the row count is only a rough estimate used in SQL optimization. (This is also true if the InnoDB table is partitioned.)

• AVG\_ROW\_LENGTH The average row length.

• DATA\_LENGTH

For MyISAM, DATA\_LENGTH is the length of the data file, in bytes.

For InnoDB, DATA\_LENGTH is the approximate amount of space allocated for the clustered index, in bytes. Specifically, it is the clustered index size, in pages, multiplied by the InnoDB page size.

Refer to the notes at the end of this section for information regarding other storage engines.

• MAX\_DATA\_LENGTH

For MyISAM, MAX\_DATA\_LENGTH is maximum length of the data file. This is the total number of bytes of data that can be stored in the table, given the data pointer size used.

Unused for InnoDB.

Refer to the notes at the end of this section for information regarding other storage engines.

• INDEX\_LENGTH

For MyISAM, INDEX\_LENGTH is the length of the index file, in bytes.

For InnoDB, INDEX\_LENGTH is the approximate amount of space allocated for non-clustered indexes, in bytes. Specifically, it is the sum of non-clustered index sizes, in pages, multiplied by the InnoDB page size.

Refer to the notes at the end of this section for information regarding other storage engines.

• DATA\_FREE

The number of allocated but unused bytes.

InnoDB tables report the free space of the tablespace to which the table belongs. For a table located in the shared tablespace, this is the free space of the shared tablespace. If you are using multiple tablespaces and the table has its own tablespace, the free space is for only that table. Free space means the number of bytes in completely free extents minus a safety margin. Even if free space displays as 0, it may be possible to insert rows as long as new extents need not be allocated.

For NDB Cluster, DATA\_FREE shows the space allocated on disk for, but not used by, a Disk Data table or fragment on disk. (In-memory data resource usage is reported by the DATA\_LENGTH column.)

For partitioned tables, this value is only an estimate and may not be absolutely correct. A more accurate method of obtaining this information in such cases is to query the INFORMATION\_SCHEMA [PARTITIONS](#_bookmark3) table, as shown in this example:

SELECT SUM(DATA\_FREE)

FROM INFORMATION\_SCHEMA.PARTITIONS

WHERE TABLE\_SCHEMA = 'mydb'

AND TABLE\_NAME = 'mytable';

For more information, see [Section 26.3.21, “The INFORMATION\_SCHEMA PARTITIONS Table”](#_bookmark3) .

• AUTO\_INCREMENT

The next AUTO\_INCREMENT value.

• CREATE\_TIME

When the table was created.

• UPDATE\_TIME

When the data file was last updated. For some storage engines, this value is NULL. For example, InnoDB stores multiple tables in its system tablespace and the data file timestamp does not apply. Even with file-per-table mode with each InnoDB table in a separate .ibd file, change buffering can delay the write to the data file, so the file modification time is different from the time of the last insert, update, or delete. For MyISAM, the data file timestamp is used; however, on Windows the timestamp is not updated by updates, so the value is inaccurate.

UPDATE\_TIME displays a timestamp value for the last UPDATE, INSERT, or DELETE performed on InnoDB tables that are not partitioned. For MVCC, the timestamp value reflects the COMMIT time,

which is considered the last update time. Timestamps are not persisted when the server is restarted or when the table is evicted from the InnoDB data dictionary cache.

• CHECK\_TIME

When the table was last checked. Not all storage engines update this time, in which case, the value is always NULL.

For partitioned InnoDB tables, CHECK\_TIME is always NULL.

• TABLE\_COLLATION

The table default collation. The output does not explicitly list the table default character set, but the collation name begins with the character set name.

• CHECKSUM

The live checksum value, if any.

• CREATE\_OPTIONS

Extra options used with CREATE TABLE.

CREATE\_OPTIONS shows partitioned for a partitioned table.

Prior to MySQL 8.0.16, CREATE\_OPTIONS shows the ENCRYPTION clause specified for tables created in file-per-table tablespaces. As of MySQL 8.0.16, it shows the encryption clause for file- per-table tablespaces if the table is encrypted or if the specified encryption differs from the schema encryption. The encryption clause is not shown for tables created in general tablespaces. To identify encrypted file-per-table and general tablespaces, query the [INNODB\_TABLESPACES](#_bookmark142) ENCRYPTION column.

When creating a table with strict mode disabled, the storage engine's default row format is used if the specified row format is not supported. The actual row format of the table is reported in the ROW\_FORMAT column. CREATE\_OPTIONS shows the row format that was specified in the CREATE TABLE statement.

When altering the storage engine of a table, table options that are not applicable to the new storage engine are retained in the table definition to enable reverting the table with its previously defined options to the original storage engine, if necessary. The CREATE\_OPTIONS column may show retained options.

• TABLE\_COMMENT

The comment used when creating the table (or information as to why MySQL could not access the table information).

**Notes**

• For NDB tables, the output of this statement shows appropriate values for the AVG\_ROW\_LENGTH and DATA\_LENGTH columns, with the exception that BLOB columns are not taken into account.

• For NDB tables, DATA\_LENGTH includes data stored in main memory only; the MAX\_DATA\_LENGTH and DATA\_FREE columns apply to Disk Data.

• For NDB Cluster Disk Data tables, MAX\_DATA\_LENGTH shows the space allocated for the disk part of a Disk Data table or fragment. (In-memory data resource usage is reported by the DATA\_LENGTH column.)

• For MEMORY tables, the DATA\_LENGTH, MAX\_DATA\_LENGTH, and INDEX\_LENGTH values approximate the actual amount of allocated memory. The allocation algorithm reserves memory in large amounts to reduce the number of allocation operations.



• For views, most [TABLES](#_bookmark107) columns are 0 or NULL except that TABLE\_NAME indicates the view name, CREATE\_TIME indicates the creation time, and TABLE\_COMMENT says VIEW.

Table information is also available from the SHOW TABLE STATUS and SHOW TABLES statements. See Section 13.7.7.38, “SHOW TABLE STATUS Statement” , and Section 13.7.7.39, “SHOW TABLES Statement” . The following statements are equivalent:

SELECT

TABLE\_NAME, ENGINE, VERSION, ROW\_FORMAT, TABLE\_ROWS, AVG\_ROW\_LENGTH,

DATA\_LENGTH, MAX\_DATA\_LENGTH, INDEX\_LENGTH, DATA\_FREE, AUTO\_INCREMENT,

CREATE\_TIME, UPDATE\_TIME, CHECK\_TIME, TABLE\_COLLATION, CHECKSUM,

CREATE\_OPTIONS, TABLE\_COMMENT

FROM INFORMATION\_SCHEMA.TABLES

WHERE table\_schema = '*db\_name* '

[AND table\_name LIKE '*wild*']

SHOW TABLE STATUS

FROM *db\_name*

[LIKE '*wild*']

The following statements are equivalent:

SELECT

TABLE\_NAME, TABLE\_TYPE

FROM INFORMATION\_SCHEMA.TABLES

WHERE table\_schema = '*db\_name* '

[AND table\_name LIKE '*wild*']

SHOW FULL TABLES

FROM *db\_name*

[LIKE '*wild*']

**26.3.39** **The** **INFORMATION\_SCHEMA** **TABLES\_EXTENSIONS** **Table**

The [TABLES\_EXTENSIONS](#_bookmark108) table (available as of MySQL 8.0.21) provides information about table attributes defined for primary and secondary storage engines.

**Note**

The [TABLES\_EXTENSIONS](#_bookmark108) table is reserved for future use.

The [TABLES\_EXTENSIONS](#_bookmark108) table has these columns:

• TABLE\_CATALOG

The name of the catalog to which the table belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the table belongs.

• TABLE\_NAME The name of the table.

• ENGINE\_ATTRIBUTE

Table attributes defined for the primary storage engine. Reserved for future use.

• SECONDARY\_ENGINE\_ATTRIBUTE

Table attributes defined for the secondary storage engine. Reserved for future use.

**26.3.40** **The** **INFORMATION\_SCHEMA** **TABLESPACES** **Table**



This table is unused. It is deprecated; expect it to be removed in a future MySQL release. Other INFORMATION\_SCHEMA tables may provide related information:

• For NDB, the INFORMATION\_SCHEMA [FILES](#_bookmark86) table provides tablespace-related information.

• For InnoDB, the INFORMATION\_SCHEMA [INNODB\_TABLESPACES](#_bookmark142) and [INNODB\_DATAFILES](#_bookmark128) tables provide tablespace metadata.

**26.3.41** **The** **INFORMATION\_SCHEMA** **TABLESPACES\_EXTENSIONS** **Table**

The [TABLESPACES\_EXTENSIONS](#_bookmark110) table (available as of MySQL 8.0.21) provides information about tablespace attributes defined for primary storage engines.

**Note**

The [TABLESPACES\_EXTENSIONS](#_bookmark110) table is reserved for future use.

The [TABLESPACES\_EXTENSIONS](#_bookmark110) table has these columns:

• TABLESPACE\_NAME The name of the tablespace.

• ENGINE\_ATTRIBUTE

Tablespace attributes defined for the primary storage engine. Reserved for future use.

**26.3.42** **The** **INFORMATION\_SCHEMA** **TABLE\_CONSTRAINTS** **Table**

The [TABLE\_CONSTRAINTS](#_bookmark111) table describes which tables have constraints.

The [TABLE\_CONSTRAINTS](#_bookmark111) table has these columns:

• CONSTRAINT\_CATALOG

The name of the catalog to which the constraint belongs. This value is always def.

• CONSTRAINT\_SCHEMA

The name of the schema (database) to which the constraint belongs.

• TABLE\_SCHEMA

The name of the schema (database) to which the table belongs.

• TABLE\_NAME The name of the table.

• The CONSTRAINT\_TYPE

The type of constraint. The value can be UNIQUE, PRIMARY KEY, FOREIGN KEY, or (as of MySQL 8.0.16) CHECK. This is a CHAR (not ENUM) column.

The UNIQUE and PRIMARY KEY information is about the same as what you get from the Key\_name column in the output from SHOW INDEX when the Non\_unique column is 0.

• ENFORCED

For CHECK constraints, the value is YES or NO to indicate whether the constraint is enforced. For

other constraints, the value is always YES.

This column was added in MySQL 8.0.16.



**26.3.43** **The** **INFORMATION\_SCHEMA** **TABLE\_CONSTRAINTS\_EXTENSIONS**

**Table**

The [TABLE\_CONSTRAINTS\_EXTENSIONS](#_bookmark112) table (available as of MySQL 8.0.21) provides information about table constraint attributes defined for primary and secondary storage engines.

**Note**

The [TABLE\_CONSTRAINTS\_EXTENSIONS](#_bookmark112) table is reserved for future use.

The [TABLE\_CONSTRAINTS\_EXTENSIONS](#_bookmark112) table has these columns:

• CONSTRAINT\_CATALOG

The name of the catalog to which the table belongs.

• CONSTRAINT\_SCHEMA

The name of the schema (database) to which the table belongs.

• CONSTRAINT\_NAME The name of the constraint.

• TABLE\_NAME The name of the table.

• ENGINE\_ATTRIBUTE

Constraint attributes defined for the primary storage engine. Reserved for future use.

• SECONDARY\_ENGINE\_ATTRIBUTE

Constraint attributes defined for the secondary storage engine. Reserved for future use.

**26.3.44** **The** **INFORMATION\_SCHEMA** **TABLE\_PRIVILEGES** **Table**

The [TABLE\_PRIVILEGES](#_bookmark113) table provides information about table privileges. It takes its values from the mysql.tables\_priv system table.

The [TABLE\_PRIVILEGES](#_bookmark113) table has these columns:

• GRANTEE

The name of the account to which the privilege is granted, in '*user\_name*'@'*host\_name*' format.

• TABLE\_CATALOG

The name of the catalog to which the table belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the table belongs.

• TABLE\_NAME The name of the table.

• PRIVILEGE\_TYPE

The privilege granted. The value can be any privilege that can be granted at the table level; see Section 13.7.1.6, “GRANT Statement” . Each row lists a single privilege, so there is one row per table privilege held by the grantee.

• IS\_GRANTABLE

YES if the user has the GRANT OPTION privilege, NO otherwise. The output does not list GRANT OPTION as a separate row with PRIVILEGE\_TYPE='GRANT OPTION'.

**Notes**

• [TABLE\_PRIVILEGES](#_bookmark113) is a nonstandard INFORMATION\_SCHEMA table.

The following statements are *not* equivalent:

SELECT ... FROM INFORMATION\_SCHEMA.TABLE\_PRIVILEGES

SHOW GRANTS ...

**26.3.45** **The** **INFORMATION\_SCHEMA** **TRIGGERS** **Table**

The [TRIGGERS](#_bookmark52) table provides information about triggers. To see information about a table's triggers, you must have the TRIGGER privilege for the table.

The [TRIGGERS](#_bookmark52) table has these columns:

• TRIGGER\_CATALOG

The name of the catalog to which the trigger belongs. This value is always def.

• TRIGGER\_SCHEMA

The name of the schema (database) to which the trigger belongs.

• TRIGGER\_NAME The name of the trigger.

• EVENT\_MANIPULATION

The trigger event. This is the type of operation on the associated table for which the trigger activates. The value is INSERT (a row was inserted), DELETE (a row was deleted), or UPDATE (a row was modified).

• EVENT\_OBJECT\_CATALOG, EVENT\_OBJECT\_SCHEMA, and EVENT\_OBJECT\_TABLE

As noted in [Section 25.3, “Using Triggers”](#_bookmark30) , every trigger is associated with exactly one table. These columns indicate the catalog and schema (database) in which this table occurs, and the table name, respectively. The EVENT\_OBJECT\_CATALOG value is always def.

• ACTION\_ORDER

The ordinal position of the trigger's action within the list of triggers on the same table with the same EVENT\_MANIPULATION and ACTION\_TIMING values.

• ACTION\_CONDITION

This value is always NULL.

• ACTION\_STATEMENT

The trigger body; that is, the statement executed when the trigger activates. This text uses UTF-8 encoding.

• ACTION\_ORIENTATION This value is always ROW.

• ACTION\_TIMING

Whether the trigger activates before or after the triggering event. The value is BEFORE or AFTER.

• ACTION\_REFERENCE\_OLD\_TABLE This value is always NULL.

• ACTION\_REFERENCE\_NEW\_TABLE This value is always NULL.

• ACTION\_REFERENCE\_OLD\_ROW and ACTION\_REFERENCE\_NEW\_ROW

The old and new column identifiers, respectively. The ACTION\_REFERENCE\_OLD\_ROW value is always OLD and the ACTION\_REFERENCE\_NEW\_ROW value is always NEW.

• CREATED

The date and time when the trigger was created. This is a TIMESTAMP(2) value (with a fractional part in hundredths of seconds) for triggers.

• SQL\_MODE

The SQL mode in effect when the trigger was created, and under which the trigger executes. For the permitted values, see Section 5.1.11, “Server SQL Modes” .

• DEFINER

The account named in the DEFINER clause (often the user who created the trigger), in '*user\_name*'@'*host\_name*' format.

• CHARACTER\_SET\_CLIENT

The session value of the character\_set\_client system variable when the trigger was created.

• COLLATION\_CONNECTION

The session value of the collation\_connection system variable when the trigger was created.

• DATABASE\_COLLATION

The collation of the database with which the trigger is associated.

**Example**

The following example uses the ins\_sum trigger defined in [Section 25.3, “Using Triggers”](#_bookmark30):

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.TRIGGERS**

**WHERE** **TRIGGER\_SCHEMA='test'** **AND** **TRIGGER\_NAME='ins\_sum'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TRIGGER\_CATALOG: def

TRIGGER\_SCHEMA: test

TRIGGER\_NAME: ins\_sum

EVENT\_MANIPULATION: INSERT

EVENT\_OBJECT\_CATALOG: def

EVENT\_OBJECT\_SCHEMA: test

EVENT\_OBJECT\_TABLE: account

ACTION\_ORDER: 1

ACTION\_CONDITION: NULL

ACTION\_STATEMENT: SET @sum = @sum + NEW.amount

ACTION\_ORIENTATION: ROW

ACTION\_TIMING: BEFORE

ACTION\_REFERENCE\_OLD\_TABLE: NULL

ACTION\_REFERENCE\_NEW\_TABLE: NULL

ACTION\_REFERENCE\_OLD\_ROW: OLD

ACTION\_REFERENCE\_NEW\_ROW: NEW

CREATED: 2018-08-08 10:10:12.61

SQL\_MODE: ONLY\_FULL\_GROUP\_BY,STRICT\_TRANS\_TABLES,

NO\_ZERO\_IN\_DATE,NO\_ZERO\_DATE,

ERROR\_FOR\_DIVISION\_BY\_ZERO,

NO\_ENGINE\_SUBSTITUTION

DEFINER: me@localhost

CHARACTER\_SET\_CLIENT: utf8mb4

COLLATION\_CONNECTION: utf8mb4\_0900\_ai\_ci

DATABASE\_COLLATION: utf8mb4\_0900\_ai\_ci

Trigger information is also available from the SHOW TRIGGERS statement. See Section 13.7.7.40,

“SHOW TRIGGERS Statement” .

**26.3.46** **The** **INFORMATION\_SCHEMA** **USER\_ATTRIBUTES** **Table**

The [USER\_ATTRIBUTES](#_bookmark114) table (available as of MySQL 8.0.21) provides information about user

comments and user attributes. It takes its values from the mysql.user system table. The [USER\_ATTRIBUTES](#_bookmark114) table has these columns:

• USER

The user name portion of the account to which the ATTRIBUTE column value applies.

• HOST

The host name portion of the account to which the ATTRIBUTE column value applies.

• ATTRIBUTE

The user comment, user attribute, or both belonging to the account specified by the USER and HOST columns. The value is in JSON object notation. Attributes are shown exactly as set using CREATE USER and ALTER USER statements with ATTRIBUTE or COMMENT options. A comment is shown as a key-value pair having comment as the key. For additional information and examples, see CREATE USER Comment and Attribute Options.

**Notes**

• [USER\_ATTRIBUTES](#_bookmark114) is a nonstandard INFORMATION\_SCHEMA table.

• To obtain only the user comment for a given user as an unquoted string, you can employ a query such as this one:

mysql> **SELECT** **ATTRIBUTE->>"$** **.comment"** **AS** **Comment**

|  |  |
| --- | --- |
| ->  -> | **FROM** **INFORMATION\_SCHEMA.USER\_ATTRIBUTES**  **WHERE** **USER='bill'** **AND** **HOST='localhost';** |

+-----------+

| Comment |

+-----------+

| A comment |

+-----------+

Similarly, you can obtain the unquoted value for a given user attribute using its key.

• Prior to MySQL 8.0.22, [USER\_ATTRIBUTES](#_bookmark114) contents are accessible by anyone. As of MySQL 8.0.22, [USER\_ATTRIBUTES](#_bookmark114) contents are accessible as follows:

• All rows are accessible if:

• The current thread is a replica thread.

• The access control system has not been initialized (for example, the server was started with the --skip-grant-tables option).

• The currently authenticated account has the UPDATE or SELECT privilege for the mysql.user system table.

• The currently authenticated account has the CREATE USER and SYSTEM\_USER privileges.

• Otherwise, the currently authenticated account can see the row for that account. Additionally, if the account has the CREATE USER privilege but not the SYSTEM\_USER privilege, it can see rows for all other accounts that do not have the SYSTEM\_USER privilege.

For more information about specifying account comments and attributes, see Section 13.7.1.3,

“CREATE USER Statement” .

**26.3.47** **The** **INFORMATION\_SCHEMA** **USER\_PRIVILEGES** **Table**

The [USER\_PRIVILEGES](#_bookmark115) table provides information about global privileges. It takes its values from the mysql.user system table.

The [USER\_PRIVILEGES](#_bookmark115) table has these columns:

• GRANTEE

The name of the account to which the privilege is granted, in '*user\_name*'@'*host\_name*' format.

• TABLE\_CATALOG

The name of the catalog. This value is always def.

• PRIVILEGE\_TYPE

The privilege granted. The value can be any privilege that can be granted at the global level; see Section 13.7.1.6, “GRANT Statement” . Each row lists a single privilege, so there is one row per global privilege held by the grantee.

• IS\_GRANTABLE

YES if the user has the GRANT OPTION privilege, NO otherwise. The output does not list GRANT OPTION as a separate row with PRIVILEGE\_TYPE='GRANT OPTION'.

**Notes**

• [USER\_PRIVILEGES](#_bookmark115) is a nonstandard INFORMATION\_SCHEMA table.

The following statements are *not* equivalent:

SELECT ... FROM INFORMATION\_SCHEMA.USER\_PRIVILEGES

SHOW GRANTS ...

**26.3.48** **The** **INFORMATION\_SCHEMA** **VIEWS** **Table**

The [VIEWS](#_bookmark54) table provides information about views in databases. You must have the SHOW VIEW privilege to access this table.

The [VIEWS](#_bookmark54) table has these columns:

• TABLE\_CATALOG

The name of the catalog to which the view belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the view belongs.

• TABLE\_NAME The name of the view.

• VIEW\_DEFINITION

The SELECT statement that provides the definition of the view. This column has most of what you see in the Create Table column that SHOW CREATE VIEW produces. Skip the words before SELECT and skip the words WITH CHECK OPTION. Suppose that the original statement was:

CREATE VIEW v AS

SELECT s2,s1 FROM t

WHERE s1 > 5

ORDER BY s1

WITH CHECK OPTION;

Then the view definition looks like this:

SELECT s2,s1 FROM t WHERE s1 > 5 ORDER BY s1

• CHECK\_OPTION

The value of the CHECK\_OPTION attribute. The value is one of NONE, CASCADE, or LOCAL.

• IS\_UPDATABLE

MySQL sets a flag, called the view updatability flag, at CREATE VIEW time. The flag is set to YES (true) if UPDATE and DELETE (and similar operations) are legal for the view. Otherwise, the flag is set to NO (false). The IS\_UPDATABLE column in the [VIEWS](#_bookmark54) table displays the status of this flag. It means that the server always knows whether a view is updatable.

If a view is not updatable, statements such UPDATE, DELETE, and INSERT are illegal and are rejected. (Even if a view is updatable, it might not be possible to insert into it; for details, refer to [Section 25.5.3, “Updatable and Insertable Views”](#_bookmark43) .)

• DEFINER

The account of the user who created the view, in '*user\_name*'@'*host\_name*' format.

• SECURITY\_TYPE

The view SQL SECURITY characteristic. The value is one of DEFINER or INVOKER.

• CHARACTER\_SET\_CLIENT

The session value of the character\_set\_client system variable when the view was created.

• COLLATION\_CONNECTION

The session value of the collation\_connection system variable when the view was created.

**Notes**

MySQL permits different sql\_mode settings to tell the server the type of SQL syntax to support. For example, you might use the ANSI SQL mode to ensure MySQL correctly interprets the standard SQL concatenation operator, the double bar (||), in your queries. If you then create a view that concatenates items, you might worry that changing the sql\_mode setting to a value different from ANSI could cause the view to become invalid. But this is not the case. No matter how you write out a view definition, MySQL always stores it the same way, in a canonical form. Here is an example that shows how the server changes a double bar concatenation operator to a CONCAT() function:

mysql> **SET** **sql\_mode** **=** **'ANSI';**

Query OK, 0 rows affected (0.00 sec)

mysql> **CREATE** **VIEW** **test** **.v** **AS** **SELECT** **'a'** **||** **'b'** **as** **col1;**

Query OK, 0 rows affected (0.00 sec)

mysql> **SELECT** **VIEW\_DEFINITION** **FROM** **INFORMATION\_SCHEMA.VIEWS**

**WHERE** **TABLE\_SCHEMA** **=** **'test'** **AND** **TABLE\_NAME** **=** **'v';**

+----------------------------------+

| VIEW\_DEFINITION |

+----------------------------------+

| select concat('a','b') AS `col1` |

+----------------------------------+

1 row in set (0.00 sec)

The advantage of storing a view definition in canonical form is that changes made later to the value of sql\_mode do not affect the results from the view. However, an additional consequence is that comments prior to SELECT are stripped from the definition by the server.

**26.3.49** **The** **INFORMATION\_SCHEMA** **VIEW\_ROUTINE\_USAGE** **Table**

The [VIEW\_ROUTINE\_USAGE](#_bookmark116) table (available as of MySQL 8.0.13) provides access to information about stored functions used in view definitions. The table does not list information about built-in (native) functions or loadable functions used in the definitions.

You can see information only for views for which you have some privilege, and only for functions for which you have some privilege.

The [VIEW\_ROUTINE\_USAGE](#_bookmark116) table has these columns:

• TABLE\_CATALOG

The name of the catalog to which the view belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the view belongs.

• TABLE\_NAME The name of the view.

• SPECIFIC\_CATALOG

The name of the catalog to which the function used in the view definition belongs. This value is always def.

• SPECIFIC\_SCHEMA

The name of the schema (database) to which the function used in the view definition belongs.

• SPECIFIC\_NAME

The name of the function used in the view definition.

**26.3.50** **The** **INFORMATION\_SCHEMA** **VIEW\_TABLE\_USAGE** **Table**

The [VIEW\_TABLE\_USAGE](#_bookmark117) table (available as of MySQL 8.0.13) provides access to information about tables and views used in view definitions.

You can see information only for views for which you have some privilege, and only for tables for which you have some privilege.

The [VIEW\_TABLE\_USAGE](#_bookmark117) table has these columns:

• VIEW\_CATALOG

The name of the catalog to which the view belongs. This value is always def.

• VIEW\_SCHEMA

The name of the schema (database) to which the view belongs.

• VIEW\_NAME The name of the view.

• TABLE\_CATALOG

The name of the catalog to which the table or view used in the view definition belongs. This value is always def.

• TABLE\_SCHEMA

The name of the schema (database) to which the table or view used in the view definition belongs.

• TABLE\_NAME

The name of the table or view used in the view definition.

**26.4** **INFORMATION\_SCHEMA** **InnoDB** **Tables**

This section provides table definitions for INFORMATION\_SCHEMA InnoDB tables. For related information and examples, see Section 15.15, “InnoDB INFORMATION\_SCHEMA Tables” .

INFORMATION\_SCHEMA InnoDB tables can be used to monitor ongoing InnoDB activity, to detect inefficiencies before they turn into issues, or to troubleshoot performance and capacity issues. As your database becomes bigger and busier, running up against the limits of your hardware capacity, you monitor and tune these aspects to keep the database running smoothly.

**26.4.1** **INFORMATION\_SCHEMA** **InnoDB** **Table** **Reference**

The following table summarizes INFORMATION\_SCHEMA InnoDB tables. For greater detail, see the individual table descriptions.

**Table** **26.3** **INFORMATION\_SCHEMA** **InnoDB** **Tables**

|  |  |  |
| --- | --- | --- |
| **Table** **Name** | **Description** | **Introduced** |
| [INNODB\_BUFFER\_PAGE](#_bookmark120) | Pages in InnoDB buffer pool |  |
| [INNODB\_BUFFER\_PAGE\_LRU](#_bookmark121) | LRU ordering of pages in InnoDB buffer pool |  |
| [INNODB\_BUFFER\_POOL\_STATS](#_bookmark122) | InnoDB buffer pool statistics |  |
| [INNODB\_CACHED\_INDEXES](#_bookmark123) | Number of index pages cached per index in InnoDB buffer pool |  |
| [INNODB\_CMP](#_bookmark124) | Status for operations related to compressed InnoDB tables |  |
| [INNODB\_CMP\_PER\_INDEX](#_bookmark126) | Status for operations related to compressed InnoDB tables and indexes |  |
| [INNODB\_CMP\_PER\_INDEX\_RESE](#_bookmark126) | TStatus for operations related to compressed InnoDB tables and indexes |  |
| [INNODB\_CMP\_RESET](#_bookmark124) | Status for operations related to compressed InnoDB tables |  |
| [INNODB\_CMPMEM](#_bookmark125) | Status for compressed pages within InnoDB buffer pool |  |
| [INNODB\_CMPMEM\_RESET](#_bookmark125) | Status for compressed pages within InnoDB buffer pool |  |

|  |  |  |
| --- | --- | --- |
| **Table** **Name** | **Description** | **Introduced** |
| [INNODB\_COLUMNS](#_bookmark127) | Columns in each InnoDB table |  |
| [INNODB\_DATAFILES](#_bookmark128) | Data file path information for InnoDB file-per-table and general tablespaces |  |
| [INNODB\_FIELDS](#_bookmark129) | Key columns of InnoDB indexes |  |
| [INNODB\_FOREIGN](#_bookmark130) | InnoDB foreign-key metadata |  |
| [INNODB\_FOREIGN\_COLS](#_bookmark131) | InnoDB foreign-key column status information |  |
| [INNODB\_FT\_BEING\_DELETED](#_bookmark132) | Snapshot of INNODB\_FT\_DELETED table |  |
| [INNODB\_FT\_CONFIG](#_bookmark133) | Metadata for InnoDB table FULLTEXT index and associated processing |  |
| [INNODB\_FT\_DEFAULT\_STOPWOR](#_bookmark134) | **D**efault list of stopwords for InnoDB FULLTEXT indexes |  |
| [INNODB\_FT\_DELETED](#_bookmark135) | Rows deleted from InnoDB table FULLTEXT index |  |
| [INNODB\_FT\_INDEX\_CACHE](#_bookmark136) | Token information for newly inserted rows in InnoDB FULLTEXT index |  |
| [INNODB\_FT\_INDEX\_TABLE](#_bookmark137) | Inverted index information for processing text searches against InnoDB table FULLTEXT index |  |
| [INNODB\_INDEXES](#_bookmark138) | InnoDB index metadata |  |
| [INNODB\_METRICS](#_bookmark139) | InnoDB performance information |  |
| INNODB\_SESSION\_TEMP\_TABLE | **S**PeAsCsEioSn temporary-tablespace metadata | 8.0.13 |
| [INNODB\_TABLES](#_bookmark141) | InnoDB table metadata |  |
| [INNODB\_TABLESPACES](#_bookmark142) | InnoDB file-per-table, general, and undo tablespace metadata |  |
| [INNODB\_TABLESPACES\_BRIEF](#_bookmark143) | Brief file-per-table, general, undo, and system tablespace metadata |  |
| [INNODB\_TABLESTATS](#_bookmark144) | InnoDB table low-level status information |  |
| [INNODB\_TEMP\_TABLE\_INFO](#_bookmark145) | Information about active user- created InnoDB temporary tables |  |
| [INNODB\_TRX](#_bookmark146) | Active InnoDB transaction  information |  |
| [INNODB\_VIRTUAL](#_bookmark147) | InnoDB virtual generated column metadata |  |

**26.4.2** **The** **INFORMATION\_SCHEMA** **INNODB\_BUFFER\_PAGE** **Table**

The [INNODB\_BUFFER\_PAGE](#_bookmark120) table provides information about each page in the InnoDB buffer pool.

For related usage information and examples, see Section 15.15.5, “InnoDB INFORMATION\_SCHEMA

Buffer Pool Tables” .



**Warning**

Querying the [INNODB\_BUFFER\_PAGE](#_bookmark120) table can affect performance. Do not query this table on a production system unless you are aware of the performance impact and have determined it to be acceptable. To avoid impacting performance on a production system, reproduce the issue you want to investigate and query buffer pool statistics on a test instance.

The [INNODB\_BUFFER\_PAGE](#_bookmark120) table has these columns:

• POOL\_ID

The buffer pool ID. This is an identifier to distinguish between multiple buffer pool instances.

• BLOCK\_ID

The buffer pool block ID.

• SPACE

The tablespace ID; the same value as INNODB\_TABLES.SPACE.

• PAGE\_NUMBER The page number.

• PAGE\_TYPE

The page type. The following table shows the permitted values.

**Table** **26.4** **INNODB\_BUFFER\_PAGE.PAGE\_TYPE** **Values**

|  |  |
| --- | --- |
| **Page** **Type** | **Description** |
| ALLOCATED | Freshly allocated page |
| BLOB | Uncompressed BLOB page |
| COMPRESSED\_BLOB2 | Subsequent comp BLOB page |
| COMPRESSED\_BLOB | First compressed BLOB page |
| ENCRYPTED\_RTREE | Encrypted R-tree |
| EXTENT\_DESCRIPTOR | Extent descriptor page |
| FILE\_SPACE\_HEADER | File space header |
| FIL\_PAGE\_TYPE\_UNUSED | Unused |
| IBUF\_BITMAP | Insert buffer bitmap |
| IBUF\_FREE\_LIST | Insert buffer free list |
| IBUF\_INDEX | Insert buffer index |
| INDEX | B-tree node |
| INODE | Index node |
| LOB\_DATA | Uncompressed LOB data |
| LOB\_FIRST | First page of uncompressed LOB |
| LOB\_INDEX | Uncompressed LOB index |
| PAGE\_IO\_COMPRESSED | Compressed page |
| PAGE\_IO\_COMPRESSED\_ENCRYPTED | Compressed and encrypted page |
| PAGE\_IO\_ENCRYPTED | Encrypted page |
| RSEG\_ARRAY | Rollback segment array |

|  |  |
| --- | --- |
| **Page** **Type** | **Description** |
| RTREE\_INDEX | R-tree index |
| SDI\_BLOB | Uncompressed SDI BLOB |
| SDI\_COMPRESSED\_BLOB | Compressed SDI BLOB |
| SDI\_INDEX | SDI index |
| SYSTEM | System page |
| TRX\_SYSTEM | Transaction system data |
| UNDO\_LOG | Undo log page |
| UNKNOWN | Unknown |
| ZLOB\_DATA | Compressed LOB data |
| ZLOB\_FIRST | First page of compressed LOB |
| ZLOB\_FRAG | Compressed LOB fragment |
| ZLOB\_FRAG\_ENTRY | Compressed LOB fragment index |
| ZLOB\_INDEX | Compressed LOB index |

• FLUSH\_TYPE

The flush type.

• FIX\_COUNT

The number of threads using this block within the buffer pool. When zero, the block is eligible to be evicted.

• IS\_HASHED

Whether a hash index has been built on this page.

• NEWEST\_MODIFICATION

The Log Sequence Number of the youngest modification.

• OLDEST\_MODIFICATION

The Log Sequence Number of the oldest modification.

• ACCESS\_TIME

An abstract number used to judge the first access time of the page.

• TABLE\_NAME

The name of the table the page belongs to. This column is applicable only to pages with a PAGE\_TYPE value of INDEX. The column is NULL if the server has not yet accessed the table.

• INDEX\_NAME

The name of the index the page belongs to. This can be the name of a clustered index or a secondary index. This column is applicable only to pages with a PAGE\_TYPE value of INDEX.

• NUMBER\_RECORDS

The number of records within the page.

• DATA\_SIZE

The sum of the sizes of the records. This column is applicable only to pages with a PAGE\_TYPE value of INDEX.

• COMPRESSED\_SIZE

The compressed page size. NULL for pages that are not compressed.

• PAGE\_STATE

The page state. The following table shows the permitted values.

**Table** **26.5** **INNODB\_BUFFER\_PAGE.PAGE\_STATE** **Values**

|  |  |
| --- | --- |
| **Page** **State** | **Description** |
| FILE\_PAGE | A buffered file page |
| MEMORY | Contains a main memory object |
| NOT\_USED | In the free list |
| NULL | Clean compressed pages, compressed pages in the flush list, pages used as buffer pool watch sentinels |
| READY\_FOR\_USE | A free page |
| REMOVE\_HASH | Hash index should be removed before placing in the free list |

• IO\_FIX

Whether any I/O is pending for this page: IO\_NONE = no pending I/O, IO\_READ = read pending, IO\_WRITE = write pending, IO\_PIN = relocation and removal from the flush not permitted.

• IS\_OLD

Whether the block is in the sublist of old blocks in the LRU list.

• FREE\_PAGE\_CLOCK

The value of the freed\_page\_clock counter when the block was the last placed at the head of the LRU list. The freed\_page\_clock counter tracks the number of blocks removed from the end of the LRU list.

• IS\_STALE

Whether the page is stale. Added in MySQL 8.0.24.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_BUFFER\_PAGE** **LIMIT** **1\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

POOL\_ID: 0

BLOCK\_ID: 0

SPACE: 97

PAGE\_NUMBER: 2473

PAGE\_TYPE: INDEX

FLUSH\_TYPE: 1

FIX\_COUNT: 0

IS\_HASHED: YES

NEWEST\_MODIFICATION: 733855581

OLDEST\_MODIFICATION: 0

ACCESS\_TIME: 3378385672

TABLE\_NAME: `employees`.`salaries`

INDEX\_NAME: PRIMARY

NUMBER\_RECORDS: 468



DATA\_SIZE: 14976

COMPRESSED\_SIZE: 0

PAGE\_STATE: FILE\_PAGE

IO\_FIX: IO\_NONE

IS\_OLD: YES

FREE\_PAGE\_CLOCK: 66

IS\_STALE: NO

**Notes**

• This table is useful primarily for expert-level performance monitoring, or when developing performance-related extensions for MySQL.

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

• When tables, table rows, partitions, or indexes are deleted, associated pages remain in the buffer pool until space is required for other data. The [INNODB\_BUFFER\_PAGE](#_bookmark120) table reports information about these pages until they are evicted from the buffer pool. For more information about how the

InnoDB manages buffer pool data, see Section 15.5. 1, “Buffer Pool” .

**26.4.3** **The** **INFORMATION\_SCHEMA** **INNODB\_BUFFER\_PAGE\_LRU** **Table**

The [INNODB\_BUFFER\_PAGE\_LRU](#_bookmark121) table provides information about the pages in the InnoDB buffer pool; in particular, how they are ordered in the LRU list that determines which pages to evict from the buffer pool when it becomes full.

The [INNODB\_BUFFER\_PAGE\_LRU](#_bookmark121) table has the same columns as the [INNODB\_BUFFER\_PAGE](#_bookmark120) table with a few exceptions. It has LRU\_POSITION and COMPRESSED columns instead of BLOCK\_ID and PAGE\_STATE columns, and it does not include and IS\_STALE column.

For related usage information and examples, see Section 15.15.5, “InnoDB INFORMATION\_SCHEMA

Buffer Pool Tables” .

**Warning**

Querying the [INNODB\_BUFFER\_PAGE\_LRU](#_bookmark121) table can affect performance. Do not query this table on a production system unless you are aware of the performance impact and have determined it to be acceptable. To avoid impacting performance on a production system, reproduce the issue you want to investigate and query buffer pool statistics on a test instance.

The [INNODB\_BUFFER\_PAGE\_LRU](#_bookmark121) table has these columns:

• POOL\_ID

The buffer pool ID. This is an identifier to distinguish between multiple buffer pool instances.

• LRU\_POSITION

The position of the page in the LRU list.

• SPACE

The tablespace ID; the same value as INNODB\_TABLES.SPACE.

• PAGE\_NUMBER The page number.

• PAGE\_TYPE

The page type. The following table shows the permitted values.

**Table** **26.6** **INNODB\_BUFFER\_PAGE\_LRU.PAGE\_TYPE** **Values**

|  |  |
| --- | --- |
| **Page** **Type** | **Description** |
| ALLOCATED | Freshly allocated page |
| BLOB | Uncompressed BLOB page |
| COMPRESSED\_BLOB2 | Subsequent comp BLOB page |
| COMPRESSED\_BLOB | First compressed BLOB page |
| ENCRYPTED\_RTREE | Encrypted R-tree |
| EXTENT\_DESCRIPTOR | Extent descriptor page |
| FILE\_SPACE\_HEADER | File space header |
| FIL\_PAGE\_TYPE\_UNUSED | Unused |
| IBUF\_BITMAP | Insert buffer bitmap |
| IBUF\_FREE\_LIST | Insert buffer free list |
| IBUF\_INDEX | Insert buffer index |
| INDEX | B-tree node |
| INODE | Index node |
| LOB\_DATA | Uncompressed LOB data |
| LOB\_FIRST | First page of uncompressed LOB |
| LOB\_INDEX | Uncompressed LOB index |
| PAGE\_IO\_COMPRESSED | Compressed page |
| PAGE\_IO\_COMPRESSED\_ENCRYPTED | Compressed and encrypted page |
| PAGE\_IO\_ENCRYPTED | Encrypted page |
| RSEG\_ARRAY | Rollback segment array |
| RTREE\_INDEX | R-tree index |
| SDI\_BLOB | Uncompressed SDI BLOB |
| SDI\_COMPRESSED\_BLOB | Compressed SDI BLOB |
| SDI\_INDEX | SDI index |
| SYSTEM | System page |
| TRX\_SYSTEM | Transaction system data |
| UNDO\_LOG | Undo log page |
| UNKNOWN | Unknown |
| ZLOB\_DATA | Compressed LOB data |
| ZLOB\_FIRST | First page of compressed LOB |
| ZLOB\_FRAG | Compressed LOB fragment |
| ZLOB\_FRAG\_ENTRY | Compressed LOB fragment index |
| ZLOB\_INDEX | Compressed LOB index |

FLUSH\_TYPE

•

The flush type.

FIX\_COUNT

•

The number of threads using this block within the buffer pool. When zero, the block is eligible to be evicted.

• IS\_HASHED

Whether a hash index has been built on this page.

• NEWEST\_MODIFICATION

The Log Sequence Number of the youngest modification.

• OLDEST\_MODIFICATION

The Log Sequence Number of the oldest modification.

• ACCESS\_TIME

An abstract number used to judge the first access time of the page.

• TABLE\_NAME

The name of the table the page belongs to. This column is applicable only to pages with a PAGE\_TYPE value of INDEX. The column is NULL if the server has not yet accessed the table.

• INDEX\_NAME

The name of the index the page belongs to. This can be the name of a clustered index or a secondary index. This column is applicable only to pages with a PAGE\_TYPE value of INDEX.

• NUMBER\_RECORDS

The number of records within the page.

• DATA\_SIZE

The sum of the sizes of the records. This column is applicable only to pages with a PAGE\_TYPE value of INDEX.

• COMPRESSED\_SIZE

The compressed page size. NULL for pages that are not compressed.

• COMPRESSED

Whether the page is compressed.

• IO\_FIX

Whether any I/O is pending for this page: IO\_NONE = no pending I/O, IO\_READ = read pending, IO\_WRITE = write pending.

• IS\_OLD

Whether the block is in the sublist of old blocks in the LRU list.

• FREE\_PAGE\_CLOCK

The value of the freed\_page\_clock counter when the block was the last placed at the head of the LRU list. The freed\_page\_clock counter tracks the number of blocks removed from the end of the LRU list.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_BUFFER\_PAGE\_LRU** **LIMIT** **1\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

POOL\_ID: 0

LRU\_POSITION: 0

SPACE: 97

PAGE\_NUMBER: 1984

PAGE\_TYPE: INDEX

FLUSH\_TYPE: 1

FIX\_COUNT: 0

IS\_HASHED: YES

NEWEST\_MODIFICATION: 719490396

OLDEST\_MODIFICATION: 0

ACCESS\_TIME: 3378383796

TABLE\_NAME: `employees`.`salaries`

INDEX\_NAME: PRIMARY

NUMBER\_RECORDS: 468

DATA\_SIZE: 14976

COMPRESSED\_SIZE: 0

COMPRESSED: NO

IO\_FIX: IO\_NONE

IS\_OLD: YES

FREE\_PAGE\_CLOCK: 0

**Notes**

• This table is useful primarily for expert-level performance monitoring, or when developing performance-related extensions for MySQL.

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

• Querying this table can require MySQL to allocate a large block of contiguous memory, more than 64 bytes times the number of active pages in the buffer pool. This allocation could potentially cause an out-of-memory error, especially for systems with multi-gigabyte buffer pools.

• Querying this table requires MySQL to lock the data structure representing the buffer pool while traversing the LRU list, which can reduce concurrency, especially for systems with multi-gigabyte buffer pools.

• When tables, table rows, partitions, or indexes are deleted, associated pages remain in the buffer pool until space is required for other data. The [INNODB\_BUFFER\_PAGE\_LRU](#_bookmark121) table reports

information about these pages until they are evicted from the buffer pool. For more information about how the InnoDB manages buffer pool data, see Section 15.5. 1, “Buffer Pool” .

**26.4.4** **The** **INFORMATION\_SCHEMA** **INNODB\_BUFFER\_POOL\_STATS** **Table**

The [INNODB\_BUFFER\_POOL\_STATS](#_bookmark122) table provides much of the same buffer pool information provided in SHOW ENGINE INNODB STATUS output. Much of the same information may also be obtained using InnoDB buffer pool server status variables.

The idea of making pages in the buffer pool “young” or “not young” refers to transferring them between the sublists at the head and tail of the buffer pool data structure. Pages made “young” take longer to age out of the buffer pool, while pages made “not young” are moved much closer to the point of eviction.

For related usage information and examples, see Section 15.15.5, “InnoDB INFORMATION\_SCHEMA

Buffer Pool Tables” .

The [INNODB\_BUFFER\_POOL\_STATS](#_bookmark122) table has these columns:

• POOL\_ID

The buffer pool ID. This is an identifier to distinguish between multiple buffer pool instances.

• POOL\_SIZE

The InnoDB buffer pool size in pages.

• FREE\_BUFFERS

The number of free pages in the InnoDB buffer pool.

• DATABASE\_PAGES

The number of pages in the InnoDB buffer pool containing data. This number includes both dirty and clean pages.

• OLD\_DATABASE\_PAGES

The number of pages in the old buffer pool sublist.

• MODIFIED\_DATABASE\_PAGES

The number of modified (dirty) database pages.

• PENDING\_DECOMPRESS

The number of pages pending decompression.

• PENDING\_READS

The number of pending reads.

• PENDING\_FLUSH\_LRU

The number of pages pending flush in the LRU.

• PENDING\_FLUSH\_LIST

The number of pages pending flush in the flush list.

• PAGES\_MADE\_YOUNG

The number of pages made young.

• PAGES\_NOT\_MADE\_YOUNG The number of pages not made young.

• PAGES\_MADE\_YOUNG\_RATE

The number of pages made young per second (pages made young since the last printout / time elapsed).

• PAGES\_MADE\_NOT\_YOUNG\_RATE

The number of pages not made per second (pages not made young since the last printout / time elapsed).

• NUMBER\_PAGES\_READ The number of pages read.

• NUMBER\_PAGES\_CREATED The number of pages created.

• NUMBER\_PAGES\_WRITTEN The number of pages written.

• PAGES\_READ\_RATE

The number of pages read per second (pages read since the last printout / time elapsed).

• PAGES\_CREATE\_RATE

The number of pages created per second (pages created since the last printout / time elapsed).

• PAGES\_WRITTEN\_RATE

The number of pages written per second (pages written since the last printout / time elapsed).

• NUMBER\_PAGES\_GET

The number of logical read requests.

• HIT\_RATE

The buffer pool hit rate.

• YOUNG\_MAKE\_PER\_THOUSAND\_GETS

The number of pages made young per thousand gets.

• NOT\_YOUNG\_MAKE\_PER\_THOUSAND\_GETS

The number of pages not made young per thousand gets.

• NUMBER\_PAGES\_READ\_AHEAD The number of pages read ahead.

• NUMBER\_READ\_AHEAD\_EVICTED

The number of pages read into the InnoDB buffer pool by the read-ahead background thread that were subsequently evicted without having been accessed by queries.

• READ\_AHEAD\_RATE

The read-ahead rate per second (pages read ahead since the last printout / time elapsed).

• READ\_AHEAD\_EVICTED\_RATE

The number of read-ahead pages evicted without access per second (read-ahead pages not accessed since the last printout / time elapsed).

• LRU\_IO\_TOTAL

Total LRU I/O.

• LRU\_IO\_CURRENT

LRU I/O for the current interval.

• UNCOMPRESS\_TOTAL

The total number of pages decompressed.

• UNCOMPRESS\_CURRENT

The number of pages decompressed in the current interval.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_BUFFER\_POOL\_STATS\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

POOL\_ID: 0

POOL\_SIZE: 8192

FREE\_BUFFERS: 1

DATABASE\_PAGES: 8085

OLD\_DATABASE\_PAGES: 2964

MODIFIED\_DATABASE\_PAGES: 0

PENDING\_DECOMPRESS: 0

PENDING\_READS: 0

PENDING\_FLUSH\_LRU: 0

PENDING\_FLUSH\_LIST: 0

PAGES\_MADE\_YOUNG: 22821

PAGES\_NOT\_MADE\_YOUNG: 3544303

PAGES\_MADE\_YOUNG\_RATE: 357.62602199870594

PAGES\_MADE\_NOT\_YOUNG\_RATE: 0

NUMBER\_PAGES\_READ: 2389

NUMBER\_PAGES\_CREATED: 12385

NUMBER\_PAGES\_WRITTEN: 13111

PAGES\_READ\_RATE: 0

PAGES\_CREATE\_RATE: 0

PAGES\_WRITTEN\_RATE: 0

NUMBER\_PAGES\_GET: 33322210

HIT\_RATE: 1000

YOUNG\_MAKE\_PER\_THOUSAND\_GETS: 18

NOT\_YOUNG\_MAKE\_PER\_THOUSAND\_GETS: 0

NUMBER\_PAGES\_READ\_AHEAD: 2024

NUMBER\_READ\_AHEAD\_EVICTED: 0

READ\_AHEAD\_RATE: 0

READ\_AHEAD\_EVICTED\_RATE: 0

LRU\_IO\_TOTAL: 0

LRU\_IO\_CURRENT: 0

UNCOMPRESS\_TOTAL: 0

UNCOMPRESS\_CURRENT: 0

**Notes**

• This table is useful primarily for expert-level performance monitoring, or when developing performance-related extensions for MySQL.

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.5** **The** **INFORMATION\_SCHEMA** **INNODB\_CACHED\_INDEXES** **Table**

The [INNODB\_CACHED\_INDEXES](#_bookmark123) table reports the number of index pages cached in the InnoDB buffer pool for each index.

For related usage information and examples, see Section 15.15.5, “InnoDB INFORMATION\_SCHEMA

Buffer Pool Tables” .

The [INNODB\_CACHED\_INDEXES](#_bookmark123) table has these columns:

• SPACE\_ID The tablespace ID.

• INDEX\_ID

An identifier for the index. Index identifiers are unique across all the databases in an instance.

• N\_CACHED\_PAGES

The number of index pages cached in the InnoDB buffer pool.

**Examples**

This query returns the number of index pages cached in the InnoDB buffer pool for a specific index:

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_CACHED\_INDEXES** **WHERE** **INDEX\_ID=65\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SPACE\_ID: 4294967294

INDEX\_ID: 65

N\_CACHED\_PAGES: 45

This query returns the number of index pages cached in the InnoDB buffer pool for each index, using the [INNODB\_INDEXES](#_bookmark138) and [INNODB\_TABLES](#_bookmark141) tables to resolve the table name and index name for each INDEX\_ID value.

SELECT

tables.NAME AS table\_name,

indexes .NAME AS index\_name,

cached.N\_CACHED\_PAGES AS n\_cached\_pages

FROM

INFORMATION\_SCHEMA.INNODB\_CACHED\_INDEXES AS cached,

INFORMATION\_SCHEMA .INNODB\_INDEXES AS indexes,

INFORMATION\_SCHEMA.INNODB\_TABLES AS tables

WHERE

cached.INDEX\_ID = indexes.INDEX\_ID

AND indexes.TABLE\_ID = tables.TABLE\_ID;

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.6** **The** **INFORMATION\_SCHEMA** **INNODB\_CMP** **and**

**INNODB\_CMP\_RESET** **Tables**

The [INNODB\_CMP](#_bookmark124) and [INNODB\_CMP\_RESET](#_bookmark124) tables provide status information on operations related to compressed InnoDB tables.

The [INNODB\_CMP](#_bookmark124) and [INNODB\_CMP\_RESET](#_bookmark124) tables have these columns:

• PAGE\_SIZE

The compressed page size in bytes.

• COMPRESS\_OPS

The number of times a B-tree page of size PAGE\_SIZE has been compressed. Pages are compressed whenever an empty page is created or the space for the uncompressed modification log runs out.

• COMPRESS\_OPS\_OK

The number of times a B-tree page of size PAGE\_SIZE has been successfully compressed. This count should never exceed COMPRESS\_OPS.

• COMPRESS\_TIME

The total time in seconds used for attempts to compress B-tree pages of size PAGE\_SIZE.

• UNCOMPRESS\_OPS

The number of times a B-tree page of size PAGE\_SIZE has been uncompressed. B-tree pages are uncompressed whenever compression fails or at first access when the uncompressed page does not exist in the buffer pool.

• UNCOMPRESS\_TIME

The total time in seconds used for uncompressing B-tree pages of the size PAGE\_SIZE.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_CMP\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

page\_size: 1024

compress\_ops: 0

compress\_ops\_ok: 0

compress\_time: 0

uncompress\_ops: 0

uncompress\_time: 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

page\_size: 2048

compress\_ops: 0

compress\_ops\_ok: 0

compress\_time: 0

uncompress\_ops: 0

uncompress\_time: 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

page\_size: 4096

compress\_ops: 0

compress\_ops\_ok: 0

compress\_time: 0

uncompress\_ops: 0

uncompress\_time: 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 4. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

page\_size: 8192

compress\_ops: 86955

compress\_ops\_ok: 81182

compress\_time: 27

uncompress\_ops: 26828

uncompress\_time: 5

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 5. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

page\_size: 16384

compress\_ops: 0

compress\_ops\_ok: 0

compress\_time: 0

uncompress\_ops: 0

uncompress\_time: 0

**Notes**

• Use these tables to measure the effectiveness of InnoDB table compression in your database.

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

• For usage information, see Section 15.9.1.4, “Monitoring InnoDB Table Compression at Runtime” and Section 15.15.1.3, “Using the Compression Information Schema Tables” . For general information about InnoDB table compression, see Section 15.9, “InnoDB Table and Page Compression” .

**26.4.7** **The** **INFORMATION\_SCHEMA** **INNODB\_CMPMEM** **and**

**INNODB\_CMPMEM\_RESET** **Tables**

The [INNODB\_CMPMEM](#_bookmark125) and [INNODB\_CMPMEM\_RESET](#_bookmark125) tables provide status information on compressed pages within the InnoDB buffer pool.

The [INNODB\_CMPMEM](#_bookmark125) and [INNODB\_CMPMEM\_RESET](#_bookmark125) tables have these columns:

• PAGE\_SIZE

The block size in bytes. Each record of this table describes blocks of this size.

• BUFFER\_POOL\_INSTANCE

A unique identifier for the buffer pool instance.

• PAGES\_USED

The number of blocks of size PAGE\_SIZE that are currently in use.

• PAGES\_FREE

The number of blocks of size PAGE\_SIZE that are currently available for allocation. This column shows the external fragmentation in the memory pool. Ideally, these numbers should be at most 1.

• RELOCATION\_OPS

The number of times a block of size PAGE\_SIZE has been relocated. The buddy system can relocate the allocated “buddy neighbor” of a freed block when it tries to form a bigger freed block. Reading from the [INNODB\_CMPMEM\_RESET](#_bookmark125) table resets this count.

• RELOCATION\_TIME

The total time in microseconds used for relocating blocks of size PAGE\_SIZE. Reading from the table INNODB\_CMPMEM\_RESET resets this count.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_CMPMEM\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

page\_size: 1024

buffer\_pool\_instance: 0

pages\_used: 0

pages\_free: 0

relocation\_ops: 0

relocation\_time: 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

page\_size: 2048

buffer\_pool\_instance: 0

pages\_used: 0

pages\_free: 0

relocation\_ops: 0

relocation\_time: 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

page\_size: 4096

buffer\_pool\_instance: 0

pages\_used: 0

pages\_free: 0

relocation\_ops: 0

relocation\_time: 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 4. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

page\_size: 8192

buffer\_pool\_instance: 0

pages\_used: 7673

pages\_free: 15

relocation\_ops: 4638

relocation\_time: 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 5. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

page\_size: 16384

buffer\_pool\_instance: 0

pages\_used: 0

pages\_free: 0

relocation\_ops: 0

relocation\_time: 0

**Notes**

• Use these tables to measure the effectiveness of InnoDB table compression in your database.

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

• For usage information, see Section 15.9.1.4, “Monitoring InnoDB Table Compression at Runtime” and Section 15.15.1.3, “Using the Compression Information Schema Tables” . For general information about InnoDB table compression, see Section 15.9, “InnoDB Table and Page Compression” .

**26.4.8** **The** **INFORMATION\_SCHEMA** **INNODB\_CMP\_PER\_INDEX** **and**

**INNODB\_CMP\_PER\_INDEX\_RESET** **Tables**

The [INNODB\_CMP\_PER\_INDEX](#_bookmark126) and [INNODB\_CMP\_PER\_INDEX\_RESET](#_bookmark126) tables provide status information on operations related to compressed InnoDB tables and indexes, with separate statistics for each combination of database, table, and index, to help you evaluate the performance and usefulness of compression for specific tables.

For a compressed InnoDB table, both the table data and all the secondary indexes are compressed. In this context, the table data is treated as just another index, one that happens to contain all the columns: the clustered index.

The [INNODB\_CMP\_PER\_INDEX](#_bookmark126) and [INNODB\_CMP\_PER\_INDEX\_RESET](#_bookmark126) tables have these columns:

• DATABASE\_NAME

The schema (database) containing the applicable table.

• TABLE\_NAME

The table to monitor for compression statistics.

• INDEX\_NAME

The index to monitor for compression statistics.

• COMPRESS\_OPS

The number of compression operations attempted. Pages are compressed whenever an empty page is created or the space for the uncompressed modification log runs out.

• COMPRESS\_OPS\_OK

The number of successful compression operations. Subtract from the COMPRESS\_OPS value to get the number of compression failures. Divide by the COMPRESS\_OPS value to get the percentage of compression failures.

• COMPRESS\_TIME

The total time in seconds used for compressing data in this index.

• UNCOMPRESS\_OPS

The number of uncompression operations performed. Compressed InnoDB pages are uncompressed whenever compression fails, or the first time a compressed page is accessed in the buffer pool and the uncompressed page does not exist.

• UNCOMPRESS\_TIME

The total time in seconds used for uncompressing data in this index.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_CMP\_PER\_INDEX\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

database\_name: employees

table\_name: salaries

index\_name: PRIMARY

compress\_ops: 0

compress\_ops\_ok: 0

compress\_time: 0

uncompress\_ops: 23451

uncompress\_time: 4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

database\_name: employees

table\_name: salaries

index\_name: emp\_no

compress\_ops: 0

compress\_ops\_ok: 0

compress\_time: 0

uncompress\_ops: 1597

uncompress\_time: 0

**Notes**

• Use these tables to measure the effectiveness of InnoDB table compression for specific tables, indexes, or both.

• You must have the PROCESS privilege to query these tables.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of these tables, including data types and default values.

• Because collecting separate measurements for every index imposes substantial performance overhead, [INNODB\_CMP\_PER\_INDEX](#_bookmark126) and [INNODB\_CMP\_PER\_INDEX\_RESET](#_bookmark126) statistics are not gathered by default. You must enable the innodb\_cmp\_per\_index\_enabled system variable before performing the operations on compressed tables that you want to monitor.

• For usage information, see Section 15.9.1.4, “Monitoring InnoDB Table Compression at Runtime” and Section 15.15.1.3, “Using the Compression Information Schema Tables” . For general information about InnoDB table compression, see Section 15.9, “InnoDB Table and Page Compression” .

**26.4.9** **The** **INFORMATION\_SCHEMA** **INNODB\_COLUMNS** **Table**

The [INNODB\_COLUMNS](#_bookmark127) table provides metadata about InnoDB table columns.

For related usage information and examples, see Section 15.15.3, “InnoDB INFORMATION\_SCHEMA Schema Object Tables” .

The [INNODB\_COLUMNS](#_bookmark127) table has these columns:

• TABLE\_ID

An identifier representing the table associated with the column; the same value as

INNODB\_TABLES.TABLE\_ID.

• NAME

The name of the column. These names can be uppercase or lowercase depending on the lower\_case\_table\_names setting. There are no special system-reserved names for columns.

• POS

The ordinal position of the column within the table, starting from 0 and incrementing sequentially. When a column is dropped, the remaining columns are reordered so that the sequence has no gaps. The POS value for a virtual generated column encodes the column sequence number and ordinal position of the column. For more information, see the POS column description in [Section 26.4.29,](#_bookmark147) [“The INFORMATION\_SCHEMA INNODB\_VIRTUAL Table”](#_bookmark147) .

• MTYPE

Stands for “main type” . A numeric identifier for the column type. 1 = VARCHAR, 2 = CHAR, 3 =

FIXBINARY, 4 = BINARY, 5 = BLOB, 6 = INT, 7 = SYS\_CHILD, 8 = SYS, 9 = FLOAT, 10 = DOUBLE, 11 = DECIMAL, 12 = VARMYSQL, 13 = MYSQL, 14 = GEOMETRY.

• PRTYPE

The InnoDB “precise type” , a binary value with bits representing MySQL data type, character set code, and nullability.

• LEN

The column length, for example 4 for INT and 8 for BIGINT. For character columns in multibyte character sets, this length value is the maximum length in bytes needed to represent a definition such as VARCHAR(*N*); that is, it might be 2\**N*, 3\**N*, and so on depending on the character encoding.

• HAS\_DEFAULT

A boolean value indicating whether a column that was added instantly using ALTER TABLE ... ADD COLUMN with ALGORITHM=INSTANT has a default value. All columns added instantly have a default value, which makes this column an indicator of whether the column was added instantly.

• DEFAULT\_VALUE

The initial default value of a column that was added instantly using ALTER TABLE ... ADD COLUMN with ALGORITHM=INSTANT. If the default value is NULL or was not specified, this column reports NULL. An explicitly specified non-NULL default value is shown in an internal binary format. Subsequent modifications of the column default value do not change the value reported by this column.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_COLUMNS** **where** **TABLE\_ID** **=** **71\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TABLE\_ID: 71

NAME: col1

POS: 0

MTYPE: 6

PRTYPE: 1027

LEN: 4

HAS\_DEFAULT: 0

DEFAULT\_VALUE: NULL

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TABLE\_ID: 71

NAME: col2

POS: 1

MTYPE: 2

PRTYPE: 524542

LEN: 10

HAS\_DEFAULT: 0

DEFAULT\_VALUE: NULL

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TABLE\_ID: 71

NAME: col3

POS: 2

MTYPE: 1

PRTYPE: 524303

LEN: 10

HAS\_DEFAULT: 0

DEFAULT\_VALUE: NULL

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.



**26.4.10** **The** **INFORMATION\_SCHEMA** **INNODB\_DATAFILES** **Table**

The [INNODB\_DATAFILES](#_bookmark128) table provides data file path information for InnoDB file-per-table and general tablespaces.

For related usage information and examples, see Section 15.15.3, “InnoDB INFORMATION\_SCHEMA Schema Object Tables” .

**Note**

The INFORMATION\_SCHEMA [FILES](#_bookmark86) table reports metadata for InnoDB tablespace types including file-per-table tablespaces, general tablespaces, the system tablespace, the global temporary tablespace, and undo tablespaces.

The [INNODB\_DATAFILES](#_bookmark128) table has these columns:

• SPACE

The tablespace ID.

• PATH

The tablespace data file path. If a file-per-table tablespace is created in a location outside the MySQL data directory, the path value is a fully qualified directory path. Otherwise, the path is relative to the data directory.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_DATAFILES** **WHERE** **SPACE** **=** **57\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SPACE: [57](#_bookmark172)

PATH: ./test/t1.ibd

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.11** **The** **INFORMATION\_SCHEMA** **INNODB\_FIELDS** **Table**

The [INNODB\_FIELDS](#_bookmark129) table provides metadata about the key columns (fields) of InnoDB indexes.

For related usage information and examples, see Section 15.15.3, “InnoDB INFORMATION\_SCHEMA Schema Object Tables” .

The [INNODB\_FIELDS](#_bookmark129) table has these columns:

• INDEX\_ID

An identifier for the index associated with this key field; the same value as

INNODB\_INDEXES.INDEX\_ID.

• NAME

The name of the original column from the table; the same value as INNODB\_COLUMNS.NAME.

• POS

The ordinal position of the key field within the index, starting from 0 and incrementing sequentially. When a column is dropped, the remaining columns are reordered so that the sequence has no gaps.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_FIELDS** **WHERE** **INDEX\_ID** **=** **117\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

INDEX\_ID: 117

NAME: col1

POS: 0

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.12** **The** **INFORMATION\_SCHEMA** **INNODB\_FOREIGN** **Table**

The [INNODB\_FOREIGN](#_bookmark130) table provides metadata about InnoDB foreign keys.

For related usage information and examples, see Section 15.15.3, “InnoDB INFORMATION\_SCHEMA Schema Object Tables” .

The [INNODB\_FOREIGN](#_bookmark130) table has these columns:

• ID

The name (not a numeric value) of the foreign key index, preceded by the schema (database) name (for example, test/products\_fk).

• FOR\_NAME

The name of the child table in this foreign key relationship.

• REF\_NAME

The name of the parent table in this foreign key relationship.

• N\_COLS

The number of columns in the foreign key index.

• TYPE

A collection of bit flags with information about the foreign key column, ORed together. 0 = ON

DELETE/UPDATE RESTRICT, 1 = ON DELETE CASCADE, 2 = ON DELETE SET NULL, 4 = ON UPDATE CASCADE, 8 = ON UPDATE SET NULL, 16 = ON DELETE NO ACTION, 32 = ON UPDATE NO ACTION.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_FOREIGN\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ID: test/fk1

FOR\_NAME: test/child

REF\_NAME: test/parent

N\_COLS: 1

TYPE: 1

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.13** **The** **INFORMATION\_SCHEMA** **INNODB\_FOREIGN\_COLS** **Table**

The [INNODB\_FOREIGN\_COLS](#_bookmark131) table provides status information about InnoDB foreign key columns.

For related usage information and examples, see Section 15.15.3, “InnoDB INFORMATION\_SCHEMA Schema Object Tables” .

The [INNODB\_FOREIGN\_COLS](#_bookmark131) table has these columns:

• ID

The foreign key index associated with this index key field; the same value as INNODB\_FOREIGN.ID.

• FOR\_COL\_NAME

The name of the associated column in the child table.

• REF\_COL\_NAME

The name of the associated column in the parent table.

• POS

The ordinal position of this key field within the foreign key index, starting from 0.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_FOREIGN\_COLS** **WHERE** **ID** **=** **'test/fk1'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ID: test/fk1

FOR\_COL\_NAME: parent\_id

REF\_COL\_NAME: id

POS: 0

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.14** **The** **INFORMATION\_SCHEMA** **INNODB\_FT\_BEING\_DELETED** **Table**

The [INNODB\_FT\_BEING\_DELETED](#_bookmark132) table provides a snapshot of the [INNODB\_FT\_DELETED](#_bookmark135) table; it is used only during an OPTIMIZE TABLE maintenance operation. When OPTIMIZE TABLE is run, the [INNODB\_FT\_BEING\_DELETED](#_bookmark132) table is emptied, and DOC\_ID values are removed from the [INNODB\_FT\_DELETED](#_bookmark135) table. Because the contents of [INNODB\_FT\_BEING\_DELETED](#_bookmark132) typically have a short lifetime, this table has limited utility for monitoring or debugging. For information about running OPTIMIZE TABLE on tables with FULLTEXT indexes, see Section 12.10.6, “Fine-Tuning MySQL Full-

Text Search” .

This table is empty initially. Before querying it, set the value of the innodb\_ft\_aux\_table system

variable to the name (including the database name) of the table that contains the FULLTEXT index (for example, test/articles). The output appears similar to the example provided for the [INNODB\_FT\_DELETED](#_bookmark135) table.

For related usage information and examples, see Section 15.15.4, “InnoDB INFORMATION\_SCHEMA

FULLTEXT Index Tables” .

The [INNODB\_FT\_BEING\_DELETED](#_bookmark132) table has these columns:

• DOC\_ID

The document ID of the row that is in the process of being deleted. This value might reflect the value of an ID column that you defined for the underlying table, or it can be a sequence value generated by InnoDB when the table contains no suitable column. This value is used when you perform

text searches, to skip rows in the [INNODB\_FT\_INDEX\_TABLE](#_bookmark137) table before data for deleted rows is physically removed from the FULLTEXT index by an OPTIMIZE TABLE statement. For more information, see Optimizing InnoDB Full-Text Indexes.

**Notes**

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

• You must have the PROCESS privilege to query this table.

• For more information about InnoDB FULLTEXT search, see Section 15.6.2.4, “InnoDB Full-Text Indexes” , and Section 12.10, “Full-Text Search Functions” .

**26.4.15** **The** **INFORMATION\_SCHEMA** **INNODB\_FT\_CONFIG** **Table**

The [INNODB\_FT\_CONFIG](#_bookmark133) table provides metadata about the FULLTEXT index and associated processing for an InnoDB table.

This table is empty initially. Before querying it, set the value of the innodb\_ft\_aux\_table system variable to the name (including the database name) of the table that contains the FULLTEXT index (for example, test/articles).

For related usage information and examples, see Section 15.15.4, “InnoDB INFORMATION\_SCHEMA

FULLTEXT Index Tables” .

The [INNODB\_FT\_CONFIG](#_bookmark133) table has these columns:

• KEY

The name designating an item of metadata for an InnoDB table containing a FULLTEXT index.

The values for this column might change, depending on the needs for performance tuning and debugging for InnoDB full-text processing. The key names and their meanings include:

• optimize\_checkpoint\_limit: The number of seconds after which an OPTIMIZE TABLE run stops.

• synced\_doc\_id: The next DOC\_ID to be issued.

• stopword\_table\_name: The *database/table* name for a user-defined stopword table. The VALUE column is empty if there is no user-defined stopword table.

• use\_stopword: Indicates whether a stopword table is used, which is defined when the FULLTEXT index is created.

• VALUE

The value associated with the corresponding KEY column, reflecting some limit or current value for an aspect of a FULLTEXT index for an InnoDB table.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_FT\_CONFIG;**

+---------------------------+-------------------+

|  |  |  |  |
| --- | --- | --- | --- |
| | KEY | | | VALUE | | |

+---------------------------+-------------------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  | | optimize\_checkpoint\_limit  synced\_doc\_id  stopword\_table\_name  use\_stopword | |  |  |  | | 180  0  test/my\_stopwords  1 | |  |  |  | |

+---------------------------+-------------------+

**Notes**

• This table is intended only for internal configuration. It is not intended for statistical information purposes.

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

• For more information about InnoDB FULLTEXT search, see Section 15.6.2.4, “InnoDB Full-Text Indexes” , and Section 12.10, “Full-Text Search Functions” .

**26.4.16** **The** **INFORMATION\_SCHEMA** **INNODB\_FT\_DEFAULT\_STOPWORD**

**Table**

The [INNODB\_FT\_DEFAULT\_STOPWORD](#_bookmark134) table holds a list of stopwords that are used by default when creating a FULLTEXT index on InnoDB tables. For information about the default InnoDB stopword list and how to define your own stopword lists, see Section 12.10.4, “Full-Text Stopwords” .

For related usage information and examples, see Section 15.15.4, “InnoDB INFORMATION\_SCHEMA

FULLTEXT Index Tables” .

The [INNODB\_FT\_DEFAULT\_STOPWORD](#_bookmark134) table has these columns:

• value

A word that is used by default as a stopword for FULLTEXT indexes on InnoDB

tables. This is not used if you override the default stopword processing with either the innodb\_ft\_server\_stopword\_table or the innodb\_ft\_user\_stopword\_table system variable.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_FT\_DEFAULT\_STOPWORD;**

+-------+

| value |

+-------+

| a |

| about |

| an |

| are |

| as |

| at |

| be |

| by |

| com |

| de |

| en |

| for |

| from |

| how |

| i |

| in |

| is |

| it |

| la |

| of |

| on |

| or |

| that |

| the |

| this |

| to |

| was |

| what |

| when |

| where |

| who |

| will |

| with |

| und |

| the |

| www |

+-------+

36 rows in set (0.00 sec)

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

• For more information about InnoDB FULLTEXT search, see Section 15.6.2.4, “InnoDB Full-Text Indexes” , and Section 12.10, “Full-Text Search Functions” .

**26.4.17** **The** **INFORMATION\_SCHEMA** **INNODB\_FT\_DELETED** **Table**

The [INNODB\_FT\_DELETED](#_bookmark135) table stores rows that are deleted from the FULLTEXT index for an InnoDB table. To avoid expensive index reorganization during DML operations for an InnoDB FULLTEXT index, the information about newly deleted words is stored separately, filtered out of search results when you do a text search, and removed from the main search index only when you issue an OPTIMIZE TABLE statement for the InnoDB table. For more information, see Optimizing InnoDB Full-

Text Indexes.

This table is empty initially. Before querying it, set the value of the innodb\_ft\_aux\_table system variable to the name (including the database name) of the table that contains the FULLTEXT index (for example, test/articles).

For related usage information and examples, see Section 15.15.4, “InnoDB INFORMATION\_SCHEMA

FULLTEXT Index Tables” .

The [INNODB\_FT\_DELETED](#_bookmark135) table has these columns:

• DOC\_ID

The document ID of the newly deleted row. This value might reflect the value of an ID column that you defined for the underlying table, or it can be a sequence value generated by InnoDB when the table contains no suitable column. This value is used when you perform text searches, to skip rows in the [INNODB\_FT\_INDEX\_TABLE](#_bookmark137) table before data for deleted rows is physically removed from the FULLTEXT index by an OPTIMIZE TABLE statement. For more information, see Optimizing InnoDB

Full-Text Indexes.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_FT\_DELETED;**

+--------+

| DOC\_ID |

+--------+

|  |  |  |
| --- | --- | --- |
| |  |  | | 6  7  8 | |  |  | |

+--------+

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

• For more information about InnoDB FULLTEXT search, see Section 15.6.2.4, “InnoDB Full-Text Indexes” , and Section 12.10, “Full-Text Search Functions” .

**26.4.18** **The** **INFORMATION\_SCHEMA** **INNODB\_FT\_INDEX\_CACHE** **Table**

The [INNODB\_FT\_INDEX\_CACHE](#_bookmark136) table provides token information about newly inserted rows in a FULLTEXT index. To avoid expensive index reorganization during DML operations, the information about newly indexed words is stored separately, and combined with the main search index only when OPTIMIZE TABLE is run, when the server is shut down, or when the cache size exceeds a limit defined by the innodb\_ft\_cache\_size or innodb\_ft\_total\_cache\_size system variable.

This table is empty initially. Before querying it, set the value of the innodb\_ft\_aux\_table system variable to the name (including the database name) of the table that contains the FULLTEXT index (for example, test/articles).

For related usage information and examples, see Section 15.15.4, “InnoDB INFORMATION\_SCHEMA

FULLTEXT Index Tables” .

The [INNODB\_FT\_INDEX\_CACHE](#_bookmark136) table has these columns:

• WORD

A word extracted from the text of a newly inserted row.

• FIRST\_DOC\_ID

The first document ID in which this word appears in the FULLTEXT index.

• LAST\_DOC\_ID

The last document ID in which this word appears in the FULLTEXT index.

• DOC\_COUNT

The number of rows in which this word appears in the FULLTEXT index. The same word can occur several times within the cache table, once for each combination of DOC\_ID and POSITION values.

• DOC\_ID

The document ID of the newly inserted row. This value might reflect the value of an ID column that you defined for the underlying table, or it can be a sequence value generated by InnoDB when the table contains no suitable column.

• POSITION

The position of this particular instance of the word within the relevant document identified by the DOC\_ID value. The value does not represent an absolute position; it is an offset added to the POSITION of the previous instance of that word.

**Notes**

• This table is empty initially. Before querying it, set the value of the innodb\_ft\_aux\_table system variable to the name (including the database name) of the table that contains the FULLTEXT

index (for example test/articles). The following example demonstrates how to use the innodb\_ft\_aux\_table system variable to show information about a FULLTEXT index for a specified table.

mysql> **USE** **test;**

mysql> **CREATE** **TABLE** **articles** **(**

**id** **INT** **UNSIGNED** **AUTO\_INCREMENT** **NOT** **NULL** **PRIMARY** **KEY,**

**title** **VARCHAR(200),**

**body** **TEXT,**

**FULLTEXT** **(title,body)**

**)** **ENGINE=InnoDB;**

mysql> **INSERT** **INTO** **articles** **(title,body)** **VALUES**

**('MySQL** **Tutorial','DBMS** **stands** **for** **DataBase** **.** **.** **.** **'),**

**('How** **To** **Use** **MySQL** **Well','After** **you** **went** **through** **a** **.** **.** **.** **'),**

**('Optimizing** **MySQL','In** **this** **tutorial** **we** **show** **.** **.** **.** **'),**

**('1001** **MySQL** **Tricks','1** **.** **Never** **run** **mysqld** **as** **root** **.** **2.** **.** **.** **.** **'),**

**('MySQL** **vs** **.** **YourSQL','In** **the** **following** **database** **comparison** **.** **.** **.** **'),**

**('MySQL** **Security','When** **configured** **properly,** **MySQL** **.** **.** **.** **');**

mysql> **SET** **GLOBAL** **innodb\_ft\_aux\_table** **=** **'test/articles';**

mysql> **SELECT** **WORD,** **DOC\_COUNT,** **DOC\_ID,** **POSITION**

**FROM** **INFORMATION\_SCHEMA.INNODB\_FT\_INDEX\_CACHE** **LIMIT** **5;**

+------------+-----------+--------+----------+

| WORD | DOC\_COUNT | DOC\_ID | POSITION |

+------------+-----------+--------+----------+

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | 1001 | | | 1 | | | 4 | | | 0 | | |
| | | after | | | 1 | | | 2 | | | 22 | | |
| | | comparison | | | 1 | | | 5 | | | 44 | | |
| | | configured | | | 1 | | | 6 | | | 20 | | |
| | | database | | | 2 | | | 1 | | | 31 | | |

+------------+-----------+--------+----------+

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

• For more information about InnoDB FULLTEXT search, see Section 15.6.2.4, “InnoDB Full-Text Indexes” , and Section 12.10, “Full-Text Search Functions” .

**26.4.19** **The** **INFORMATION\_SCHEMA** **INNODB\_FT\_INDEX\_TABLE** **Table**

The [INNODB\_FT\_INDEX\_TABLE](#_bookmark137) table provides information about the inverted index used to process text searches against the FULLTEXT index of an InnoDB table.

This table is empty initially. Before querying it, set the value of the innodb\_ft\_aux\_table system variable to the name (including the database name) of the table that contains the FULLTEXT index (for example, test/articles).

For related usage information and examples, see Section 15.15.4, “InnoDB INFORMATION\_SCHEMA

FULLTEXT Index Tables” .

The [INNODB\_FT\_INDEX\_TABLE](#_bookmark137) table has these columns:

• WORD

A word extracted from the text of the columns that are part of a FULLTEXT.

• FIRST\_DOC\_ID

The first document ID in which this word appears in the FULLTEXT index.

• LAST\_DOC\_ID

The last document ID in which this word appears in the FULLTEXT index.

• DOC\_COUNT

The number of rows in which this word appears in the FULLTEXT index. The same word can occur several times within the cache table, once for each combination of DOC\_ID and POSITION values.

• DOC\_ID

The document ID of the row containing the word. This value might reflect the value of an ID column that you defined for the underlying table, or it can be a sequence value generated by InnoDB when the table contains no suitable column.

• POSITION

The position of this particular instance of the word within the relevant document identified by the DOC\_ID value.

**Notes**

• This table is empty initially. Before querying it, set the value of the innodb\_ft\_aux\_table system

variable to the name (including the database name) of the table that contains the FULLTEXT index (for example, test/articles). The following example demonstrates how to use the innodb\_ft\_aux\_table system variable to show information about a FULLTEXT index for a specified table. Before information for newly inserted rows appears in INNODB\_FT\_INDEX\_TABLE, the FULLTEXT index cache must be flushed to disk. This is accomplished by running an OPTIMIZE TABLE operation on the indexed table with the innodb\_optimize\_fulltext\_only system variable enabled. (The example disables that variable again at the end because it is intended to be enabled only temporarily.)

mysql> **USE** **test;**

mysql> **CREATE** **TABLE** **articles** **(**

**id** **INT** **UNSIGNED** **AUTO\_INCREMENT** **NOT** **NULL** **PRIMARY** **KEY,**

**title** **VARCHAR(200),**

**body** **TEXT,**

**FULLTEXT** **(title,body)**

**)** **ENGINE=InnoDB;**

mysql> **INSERT** **INTO** **articles** **(title,body)** **VALUES**

**('MySQL** **Tutorial','DBMS** **stands** **for** **DataBase** **.** **.** **.** **'),**

**('How** **To** **Use** **MySQL** **Well','After** **you** **went** **through** **a** **.** **.** **.** **'),**

**('Optimizing** **MySQL','In** **this** **tutorial** **we** **show** **.** **.** **.** **'),**

**('1001** **MySQL** **Tricks','1** **.** **Never** **run** **mysqld** **as** **root** **.** **2.** **.** **.** **.** **'),**

**('MySQL** **vs** **.** **YourSQL','In** **the** **following** **database** **comparison** **.** **.** **.** **'),**

**('MySQL** **Security','When** **configured** **properly,** **MySQL** **.** **.** **.** **');**

mysql> **SET** **GLOBAL** **innodb\_optimize\_fulltext\_only=ON;**

mysql> **OPTIMIZE** **TABLE** **articles;**

+---------------+----------+----------+----------+

| Table | Op | Msg\_type | Msg\_text |

+---------------+----------+----------+----------+

| test .articles | optimize | status | OK |

+---------------+----------+----------+----------+

mysql> **SET** **GLOBAL** **innodb\_ft\_aux\_table** **=** **'test/articles';**

mysql> **SELECT** **WORD,** **DOC\_COUNT,** **DOC\_ID,** **POSITION**

**FROM** **INFORMATION\_SCHEMA.INNODB\_FT\_INDEX\_TABLE** **LIMIT** **5;**

+------------+-----------+--------+----------+

| WORD | DOC\_COUNT | DOC\_ID | POSITION |

+------------+-----------+--------+----------+

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | 1001  after  comparison | |  |  | | 1  1  1 | |  |  | | 4  2  5 | |  |  | | 0  22  44 | |  |  | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | configured  database | |  | | 1  2 | |  | | 6  1 | |  | | 20  31 | |  | |

+------------+-----------+--------+----------+

mysql> **SET** **GLOBAL** **innodb\_optimize\_fulltext\_only=OFF;**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

• For more information about InnoDB FULLTEXT search, see Section 15.6.2.4, “InnoDB Full-Text Indexes” , and Section 12.10, “Full-Text Search Functions” .

**26.4.20** **The** **INFORMATION\_SCHEMA** **INNODB\_INDEXES** **Table**

The [INNODB\_INDEXES](#_bookmark138) table provides metadata about InnoDB indexes.

For related usage information and examples, see Section 15.15.3, “InnoDB INFORMATION\_SCHEMA Schema Object Tables” .

The [INNODB\_INDEXES](#_bookmark138) table has these columns:

• INDEX\_ID

An identifier for the index. Index identifiers are unique across all the databases in an instance.

• NAME

The name of the index. Most indexes created implicitly by InnoDB have consistent names but the index names are not necessarily unique. Examples: PRIMARY for a primary key index, GEN\_CLUST\_INDEX for the index representing a primary key when one is not specified, and ID\_IND, FOR\_IND, and REF\_IND for foreign key constraints.

• TABLE\_ID

An identifier representing the table associated with the index; the same value as

INNODB\_TABLES.TABLE\_ID.

• TYPE

A numeric value derived from bit-level information that identifies the index type. 0 = nonunique secondary index; 1 = automatically generated clustered index (GEN\_CLUST\_INDEX); 2 = unique nonclustered index; 3 = clustered index; 32 = full-text index; 64 = spatial index; 128 = secondary index on a virtual generated column.

• N\_FIELDS

The number of columns in the index key. For GEN\_CLUST\_INDEX indexes, this value is 0 because the index is created using an artificial value rather than a real table column.

• PAGE\_NO

The root page number of the index B-tree. For full-text indexes, the PAGE\_NO column is unused and set to -1 (FIL\_NULL) because the full-text index is laid out in several B-trees (auxiliary tables).

• SPACE

An identifier for the tablespace where the index resides. 0 means the InnoDB system tablespace. Any other number represents a table created with a separate .ibd file in file-per-table mode. This identifier stays the same after a TRUNCATE TABLE statement. Because all indexes for a table reside in the same tablespace as the table, this value is not necessarily unique.

• MERGE\_THRESHOLD

The merge threshold value for index pages. If the amount of data in 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 the neighboring index page. The default threshold value is 50%. For more information, see Section 15.8.11, “Configuring the Merge Threshold for Index Pages” .

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_INDEXES** **WHERE** **TABLE\_ID** **=** **34\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

INDEX\_ID: 39

NAME: GEN\_CLUST\_INDEX

TABLE\_ID: 34

TYPE: 1

N\_FIELDS: 0

PAGE\_NO: 3

SPACE: 23

MERGE\_THRESHOLD: 50

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

INDEX\_ID: 40

NAME: i1

TABLE\_ID: 34

TYPE: 0

N\_FIELDS: 1

PAGE\_NO: 4

SPACE: 23

MERGE\_THRESHOLD: 50

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.21** **The** **INFORMATION\_SCHEMA** **INNODB\_METRICS** **Table**

The [INNODB\_METRICS](#_bookmark139) table provides a wide variety of InnoDB performance information, complementing the specific focus areas of the Performance Schema tables for InnoDB. With simple queries, you can check the overall health of the system. With more detailed queries, you can diagnose issues such as performance bottlenecks, resource shortages, and application issues.

Each monitor represents a point within the InnoDB source code that is instrumented to gather counter information. Each counter can be started, stopped, and reset. You can also perform these actions for a group of counters using their common module name.

By default, relatively little data is collected. To start, stop, and reset counters, set one of the system variables innodb\_monitor\_enable, innodb\_monitor\_disable, innodb\_monitor\_reset, or innodb\_monitor\_reset\_all, using the name of the counter, the name of the module, a wildcard match for such a name using the “%” character, or the special keyword all.

For usage information, see Section 15.15.6, “InnoDB INFORMATION\_SCHEMA Metrics Table” . The [INNODB\_METRICS](#_bookmark139) table has these columns:

• NAME

A unique name for the counter.

• SUBSYSTEM

The aspect of InnoDB that the metric applies to.

• COUNT

The value since the counter was enabled.

• MAX\_COUNT

The maximum value since the counter was enabled.

• MIN\_COUNT

The minimum value since the counter was enabled.

• AVG\_COUNT

The average value since the counter was enabled.

• COUNT\_RESET

The counter value since it was last reset. (The \_RESET columns act like the lap counter on a stopwatch: you can measure the activity during some time interval, while the cumulative figures are still available in COUNT, MAX\_COUNT, and so on.)

• MAX\_COUNT\_RESET

The maximum counter value since it was last reset.

• MIN\_COUNT\_RESET

The minimum counter value since it was last reset.

• AVG\_COUNT\_RESET

The average counter value since it was last reset.

• TIME\_ENABLED The timestamp of the last start.

• TIME\_DISABLED The timestamp of the last stop.

• TIME\_ELAPSED

The elapsed time in seconds since the counter started.

• TIME\_RESET

The timestamp of the last reset.

• STATUS

Whether the counter is still running (enabled) or stopped (disabled).

• TYPE

Whether the item is a cumulative counter, or measures the current value of some resource.

• COMMENT

The counter description.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_METRICS** **WHERE** **NAME='dml\_inserts'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

NAME: dml\_inserts

SUBSYSTEM: dml

COUNT: 3

MAX\_COUNT: 3

MIN\_COUNT: NULL

AVG\_COUNT: 0.046153846153846156

COUNT\_RESET: 3

MAX\_COUNT\_RESET: 3

MIN\_COUNT\_RESET: NULL

AVG\_COUNT\_RESET: NULL

TIME\_ENABLED: 2014-12-04 14:18:28

TIME\_DISABLED: NULL

TIME\_ELAPSED: 65

TIME\_RESET: NULL

STATUS: enabled

TYPE: status\_counter

COMMENT: Number of rows inserted

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

• Transaction counter COUNT values may differ from the number of transaction events reported in Performance Schema EVENTS\_TRANSACTIONS\_SUMMARY tables. InnoDB counts only those transactions that it executes, whereas Performance Schema collects events for all non-aborted

transactions initiated by the server, including empty transactions.

**26.4.22** **The** **INFORMATION\_SCHEMA**

**INNODB\_SESSION\_TEMP\_TABLESPACES** **Table**

The [INNODB\_SESSION\_TEMP\_TABLESPACES](#_bookmark140) table provides metadata about session temporary tablespaces used for internal and user-created temporary tables. This table was added in MySQL

8.0.13.

The [INNODB\_SESSION\_TEMP\_TABLESPACES](#_bookmark140) table has these columns:

• ID

The process or session ID.

• SPACE

The tablespace ID. A range of 400 thousand space IDs is reserved for session temporary tablespaces. Session temporary tablespaces are recreated each time the server is started. Space IDs are not persisted when the server is shut down and may be reused.

• PATH

The tablespace data file path. A session temporary tablespace has an ibt file extension.

• SIZE

The size of the tablespace, in bytes.

• STATE

The state of the tablespace. ACTIVE indicates that the tablespace is currently used by a session. INACTIVE indicates that the tablespace is in the pool of available session temporary tablespaces.

• PURPOSE

The purpose of the tablespace. INTRINSIC indicates that the tablespace is used for optimized internal temporary tables use by the optimizer. SLAVE indicates that the tablespace is allocated for storing user-created temporary tables on a replication slave. USER indicates that the tablespace is used for user-created temporary tables. NONE indicates that the tablespace is not in use.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_SESSION\_TEMP\_TABLESPACES;**

+----+------------+----------------------------+-------+----------+-----------+

| ID | SPACE | PATH | SIZE | STATE | PURPOSE |

+----+------------+----------------------------+-------+----------+-----------+

| 8 | 4294566162 | ./#innodb\_temp/temp\_10 .ibt | 81920 | ACTIVE | INTRINSIC |

| 8 | 4294566161 | ./#innodb\_temp/temp\_9 .ibt | 98304 | ACTIVE | USER |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | 0 | | | 4294566153 | ./#innodb\_temp/temp\_1.ibt | | | 81920 | | | INACTIVE | | | NONE | | |
| | | 0 | | | 4294566154 | ./#innodb\_temp/temp\_2 .ibt | | | 81920 | | | INACTIVE | | | NONE | | |
| | | 0 | | | 4294566155 | ./#innodb\_temp/temp\_3 .ibt | | | 81920 | | | INACTIVE | | | NONE | | |
| | | 0 | | | 4294566156 | ./#innodb\_temp/temp\_4 .ibt | | | 81920 | | | INACTIVE | | | NONE | | |
| | | 0 | | | 4294566157 | ./#innodb\_temp/temp\_5 .ibt | | | 81920 | | | INACTIVE | | | NONE | | |
| | | 0 | | | 4294566158 | ./#innodb\_temp/temp\_6 .ibt | | | 81920 | | | INACTIVE | | | NONE | | |
| | | 0 | | | 4294566159 | ./#innodb\_temp/temp\_7 .ibt | | | 81920 | | | INACTIVE | | | NONE | | |
| | | 0 | | | 4294566160 | ./#innodb\_temp/temp\_8.ibt | | | 81920 | | | INACTIVE | | | NONE | | |

+----+------------+----------------------------+-------+----------+-----------+

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.23** **The** **INFORMATION\_SCHEMA** **INNODB\_TABLES** **Table**

The [INNODB\_TABLES](#_bookmark141) table provides metadata about InnoDB tables.

For related usage information and examples, see Section 15.15.3, “InnoDB INFORMATION\_SCHEMA Schema Object Tables” .

The [INNODB\_TABLES](#_bookmark141) table has these columns:

• TABLE\_ID

An identifier for the InnoDB table. This value is unique across all databases in the instance.

• NAME

The name of the table, preceded by the schema (database) name where appropriate (for example, test/t1). Names of databases and user tables are in the same case as they were originally defined, possibly influenced by the lower\_case\_table\_names setting.

• FLAG

A numeric value that represents bit-level information about table format and storage characteristics.

• N\_COLS

The number of columns in the table. The number reported includes three hidden columns that are created by InnoDB (DB\_ROW\_ID, DB\_TRX\_ID, and DB\_ROLL\_PTR). The number reported also includes virtual generated columns, if present.

• SPACE

An identifier for the tablespace where the table resides. 0 means the InnoDB system tablespace. Any other number represents either a file-per-table tablespace or a general tablespace. This

identifier stays the same after a TRUNCATE TABLE statement. For file-per-table tablespaces, this identifier is unique for tables across all databases in the instance.

• ROW\_FORMAT

The table's row format (Compact, Redundant, Dynamic, or Compressed).

• ZIP\_PAGE\_SIZE

The zip page size. Applies only to tables with a row format of Compressed.

• SPACE\_TYPE

The type of tablespace to which the table belongs. Possible values include System for the system tablespace, General for general tablespaces, and Single for file-per-table tablespaces. Tables assigned to the system tablespace using CREATE TABLE or ALTER TABLE TABLESPACE=innodb\_system have a SPACE\_TYPE of General. For more information, see

CREATE TABLESPACE.

• INSTANT\_COLS

The number of columns that existed before the first instant column was added using ALTER

TABLE ... ADD COLUMN with ALGORITHM=INSTANT. This column is no longer used as of MySQL 8.0.29 but continues to show information for tables with columns that were added instantly prior to MySQL 8.0.29.

• TOTAL\_ROW\_VERSIONS

The number of row versions for the table. The initial value is 0. The value is incremented by ALTER TABLE ... ALGORITHM=INSTANT operations that add or remove columns. When a table with instantly added or dropped columns is rebuilt due to a table-rebuilding ALTER TABLE or OPTIMIZE TABLE operation, the value is reset to 0. For more information, see Column Operations.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_TABLES** **WHERE** **TABLE\_ID** **=** **214\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TABLE\_ID: 1064

NAME: test/t1

FLAG: 33

N\_COLS: 6

SPACE: 3

ROW\_FORMAT: Dynamic

ZIP\_PAGE\_SIZE: 0

SPACE\_TYPE: Single

INSTANT\_COLS: 0

TOTAL\_ROW\_VERSIONS: 3

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.24** **The** **INFORMATION\_SCHEMA** **INNODB\_TABLESPACES** **Table**

The [INNODB\_TABLESPACES](#_bookmark142) table provides metadata about InnoDB file-per-table, general, and undo tablespaces.

For related usage information and examples, see Section 15.15.3, “InnoDB INFORMATION\_SCHEMA Schema Object Tables” .



**Note**

The INFORMATION\_SCHEMA [FILES](#_bookmark86) table reports metadata for InnoDB tablespace types including file-per-table tablespaces, general tablespaces, the system tablespace, the global temporary tablespace, and undo tablespaces.

The [INNODB\_TABLESPACES](#_bookmark142) table has these columns:

• SPACE

The tablespace ID.

• NAME

The schema (database) and table name.

• FLAG

A numeric value that represents bit-level information about tablespace format and storage characteristics.

• ROW\_FORMAT

The tablespace row format (Compact or Redundant, Dynamic or Compressed, or Undo). The data in this column is interpreted from the tablespace flag information that resides in the data file.

There is no way to determine from this flag information if the tablespace row format is Redundant or Compact, which is why one of the possible ROW\_FORMAT values is Compact or Redundant.

• PAGE\_SIZE

The tablespace page size. The data in this column is interpreted from the tablespace flags information that resides in the .ibd file.

• ZIP\_PAGE\_SIZE

The tablespace zip page size. The data in this column is interpreted from the tablespace flags information that resides in the .ibd file.

• SPACE\_TYPE

The type of tablespace. Possible values include General for general tablespaces, Single for file- per-table tablespaces, System for the system tablespace, and Undo for undo tablespaces.

• FS\_BLOCK\_SIZE

The file system block size, which is the unit size used for hole punching. This column pertains to the InnoDB transparent page compression feature.

• FILE\_SIZE

The apparent size of the file, which represents the maximum size of the file, uncompressed. This column pertains to the InnoDB transparent page compression feature.

• ALLOCATED\_SIZE

The actual size of the file, which is the amount of space allocated on disk. This column pertains to the InnoDB transparent page compression feature.

• AUTOEXTEND\_SIZE

The auto-extend size of the tablespace. This column was added in MySQL 8.0.23.

• SERVER\_VERSION

The MySQL version that created the tablespace, or the MySQL version into which the tablespace was imported, or the version of the last major MySQL version upgrade. The value is unchanged by a release series upgrade, such as an upgrade from MySQL 8.0.*x* to 8.0.*y*. The value can be considered a “creation” marker or “certified” marker for the tablespace.

• SPACE\_VERSION

The tablespace version, used to track changes to the tablespace format.

• ENCRYPTION

Whether the tablespace is encrypted. This column was added in MySQL 8.0.13.

• STATE

The tablespace state. This column was added in MySQL 8.0.14.

For file-per-table and general tablespaces, states include:

• normal: The tablespace is normal and active.

• discarded: The tablespace was discarded by an ALTER TABLE ... DISCARD TABLESPACE statement.

• corrupted: The tablespace is identified by InnoDB as corrupted. For undo tablespaces, states include:

• active: Rollback segments in the undo tablespace can be allocated to new transactions.

• inactive: Rollback segments in the undo tablespace are no longer used by new transactions. The truncate process is in progress. The undo tablespace was either selected by the purge thread implicitly or was made inactive by an ALTER UNDO TABLESPACE ... SET INACTIVE statement.

• empty: The undo tablespace was truncated and is no longer active. It is ready to be dropped or made active again by an ALTER UNDO TABLESPACE ... SET INACTIVE statement.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_TABLESPACES** **WHERE** **SPACE** **=** **26\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SPACE: 26

NAME: test/t1

FLAG: 0

ROW\_FORMAT: Compact or Redundant

PAGE\_SIZE: 16384

ZIP\_PAGE\_SIZE: 0

SPACE\_TYPE: Single

FS\_BLOCK\_SIZE: 4096

FILE\_SIZE: 98304

ALLOCATED\_SIZE: 65536

AUTOEXTEND\_SIZE: 0

SERVER\_VERSION: 8.0.23

SPACE\_VERSION: 1

ENCRYPTION: N

STATE: normal

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.25** **The** **INFORMATION\_SCHEMA** **INNODB\_TABLESPACES\_BRIEF**

**Table**

The [INNODB\_TABLESPACES\_BRIEF](#_bookmark143) table provides space ID, name, path, flag, and space type metadata for file-per-table, general, undo, and system tablespaces.

[INNODB\_TABLESPACES](#_bookmark142) provides the same metadata but loads more slowly because other metadata provided by the table, such as FS\_BLOCK\_SIZE, FILE\_SIZE, and ALLOCATED\_SIZE, must be loaded dynamically.

Space and path metadata is also provided by the [INNODB\_DATAFILES](#_bookmark128) table.

The [INNODB\_TABLESPACES\_BRIEF](#_bookmark143) table has these columns:

• SPACE

The tablespace ID.

• NAME

The tablespace name. For file-per-table tablespaces, the name is in the form of *schema/* *table\_name*.

• PATH

The tablespace data file path. If a file-per-table tablespace is created in a location outside the MySQL data directory, the path value is a fully qualified directory path. Otherwise, the path is relative to the data directory.

• FLAG

A numeric value that represents bit-level information about tablespace format and storage characteristics.

• SPACE\_TYPE

The type of tablespace. Possible values include General for InnoDB general tablespaces, Single for InnoDB file-per-table tablespaces, and System for the InnoDB system tablespace.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_TABLESPACES\_BRIEF** **WHERE** **SPACE** **=** **7;**

+-------+---------+---------------+-------+------------+

| SPACE | NAME | PATH | FLAG | SPACE\_TYPE |

+-------+---------+---------------+-------+------------+

| 7 | test/t1 | ./test/t1 .ibd | 16417 | Single |

+-------+---------+---------------+-------+------------+

**Notes**

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.26** **The** **INFORMATION\_SCHEMA** **INNODB\_TABLESTATS** **View**

The [INNODB\_TABLESTATS](#_bookmark144) table provides a view of low-level status information about InnoDB tables. This data is used by the MySQL optimizer to calculate which index to use when querying an InnoDB table. This information is derived from in-memory data structures rather than data stored on disk. There is no corresponding internal InnoDB system table.

InnoDB tables are represented in this view if they have been opened since the last server restart and have not aged out of the table cache. Tables for which persistent stats are available are always represented in this view.

Table statistics are updated only for DELETE or UPDATE operations that modify indexed columns. Statistics are not updated by operations that modify only nonindexed columns.

ANALYZE TABLE clears table statistics and sets the STATS\_INITIALIZED column to Uninitialized. Statistics are collected again the next time the table is accessed.

For related usage information and examples, see Section 15.15.3, “InnoDB INFORMATION\_SCHEMA Schema Object Tables” .

The [INNODB\_TABLESTATS](#_bookmark144) table has these columns:

• TABLE\_ID

An identifier representing the table for which statistics are available; the same value as

INNODB\_TABLES.TABLE\_ID.

• NAME

The name of the table; the same value as INNODB\_TABLES.NAME.

• STATS\_INITIALIZED

The value is Initialized if the statistics are already collected, Uninitialized if not.

• NUM\_ROWS

The current estimated number of rows in the table. Updated after each DML operation. The value could be imprecise if uncommitted transactions are inserting into or deleting from the table.

• CLUST\_INDEX\_SIZE

The number of pages on disk that store the clustered index, which holds the InnoDB table data in primary key order. This value might be null if no statistics are collected yet for the table.

• OTHER\_INDEX\_SIZE

The number of pages on disk that store all secondary indexes for the table. This value might be null if no statistics are collected yet for the table.

• MODIFIED\_COUNTER

The number of rows modified by DML operations, such as INSERT, UPDATE, DELETE, and also cascade operations from foreign keys. This column is reset each time table statistics are recalculated

• AUTOINC

The next number to be issued for any auto-increment-based operation. The rate at which the AUTOINC value changes depends on how many times auto-increment numbers have been requested and how many numbers are granted per request.

• REF\_COUNT

When this counter reaches zero, the table metadata can be evicted from the table cache.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_TABLESTATS** **where** **TABLE\_ID** **=** **71\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TABLE\_ID: 71

NAME: test/t1

STATS\_INITIALIZED: Initialized

NUM\_ROWS: 1

CLUST\_INDEX\_SIZE: 1

OTHER\_INDEX\_SIZE: 0

MODIFIED\_COUNTER: 1

AUTOINC: 0

REF\_COUNT: 1

**Notes**

• This table is useful primarily for expert-level performance monitoring, or when developing performance-related extensions for MySQL.

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.27** **The** **INFORMATION\_SCHEMA** **INNODB\_TEMP\_TABLE\_INFO** **Table**

The [INNODB\_TEMP\_TABLE\_INFO](#_bookmark145) table provides information about user-created InnoDB temporary tables that are active in an InnoDB instance. It does not provide information about internal InnoDB temporary tables used by the optimizer. The [INNODB\_TEMP\_TABLE\_INFO](#_bookmark145) table is created when first queried, exists only in memory, and is not persisted to disk.

For usage information and examples, see Section 15.15.7, “InnoDB INFORMATION\_SCHEMA

Temporary Table Info Table” .

The [INNODB\_TEMP\_TABLE\_INFO](#_bookmark145) table has these columns:

• TABLE\_ID

The table ID of the temporary table.

• NAME

The name of the temporary table.

• N\_COLS

The number of columns in the temporary table. The number includes three hidden columns created by InnoDB (DB\_ROW\_ID, DB\_TRX\_ID, and DB\_ROLL\_PTR).

• SPACE

The ID of the temporary tablespace where the temporary table resides.

**Example**

mysql> **CREATE** **TEMPORARY** **TABLE** **t1** **(c1** **INT** **PRIMARY** **KEY)** **ENGINE=INNODB;**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_TEMP\_TABLE\_INFO\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TABLE\_ID: 97

NAME: #sql8c88\_43\_0

N\_COLS: 4

SPACE: 76

**Notes**

• This table is useful primarily for expert-level monitoring.

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.28** **The** **INFORMATION\_SCHEMA** **INNODB\_TRX** **Table**

The [INNODB\_TRX](#_bookmark146) table provides information about every transaction currently executing inside InnoDB, including whether the transaction is waiting for a lock, when the transaction started, and the SQL statement the transaction is executing, if any.

For usage information, see Section 15.15.2.1, “Using InnoDB Transaction and Locking Information” . The [INNODB\_TRX](#_bookmark146) table has these columns:

• TRX\_ID

A unique transaction ID number, internal to InnoDB. These IDs are not created for transactions that are read only and nonlocking. For details, see Section 8.5.3, “Optimizing InnoDB Read-Only

Transactions” .

• TRX\_WEIGHT

The weight of a transaction, reflecting (but not necessarily the exact count of) the number of rows altered and the number of rows locked by the transaction. To resolve a deadlock, InnoDB selects the transaction with the smallest weight as the “victim” to roll back. Transactions that have changed nontransactional tables are considered heavier than others, regardless of the number of altered and locked rows.

• TRX\_STATE

The transaction execution state. Permitted values are RUNNING, LOCK WAIT, ROLLING BACK, and

COMMITTING.

• TRX\_STARTED The transaction start time.

• TRX\_REQUESTED\_LOCK\_ID

The ID of the lock the transaction is currently waiting for, if TRX\_STATE is LOCK WAIT; otherwise NULL. To obtain details about the lock, join this column with the ENGINE\_LOCK\_ID column of the Performance Schema data\_locks table.

• TRX\_WAIT\_STARTED

The time when the transaction started waiting on the lock, if TRX\_STATE is LOCK WAIT; otherwise

NULL.

• TRX\_MYSQL\_THREAD\_ID

The MySQL thread ID. To obtain details about the thread, join this column with the ID column of the INFORMATION\_SCHEMA [PROCESSLIST](#_bookmark93) table, but see Section 15.15.2.3, “Persistence and Consistency of InnoDB Transaction and Locking Information” .

• TRX\_QUERY

The SQL statement that is being executed by the transaction.

• TRX\_OPERATION\_STATE

The transaction's current operation, if any; otherwise NULL.

• TRX\_TABLES\_IN\_USE

The number of InnoDB tables used while processing the current SQL statement of this transaction.

• TRX\_TABLES\_LOCKED

The number of InnoDB tables that the current SQL statement has row locks on. (Because these are row locks, not table locks, the tables can usually still be read from and written to by multiple transactions, despite some rows being locked.)

• TRX\_LOCK\_STRUCTS

The number of locks reserved by the transaction.

• TRX\_LOCK\_MEMORY\_BYTES

The total size taken up by the lock structures of this transaction in memory.

• TRX\_ROWS\_LOCKED

The approximate number or rows locked by this transaction. The value might include delete-marked rows that are physically present but not visible to the transaction.

• TRX\_ROWS\_MODIFIED

The number of modified and inserted rows in this transaction.

• TRX\_CONCURRENCY\_TICKETS

A value indicating how much work the current transaction can do before being swapped out, as specified by the innodb\_concurrency\_tickets system variable.

• TRX\_ISOLATION\_LEVEL

The isolation level of the current transaction.

• TRX\_UNIQUE\_CHECKS

Whether unique checks are turned on or off for the current transaction. For example, they might be turned off during a bulk data load.

• TRX\_FOREIGN\_KEY\_CHECKS

Whether foreign key checks are turned on or off for the current transaction. For example, they might be turned off during a bulk data load.

• TRX\_LAST\_FOREIGN\_KEY\_ERROR

The detailed error message for the last foreign key error, if any; otherwise NULL.

• TRX\_ADAPTIVE\_HASH\_LATCHED

Whether the adaptive hash index is locked by the current transaction. When the adaptive hash index search system is partitioned, a single transaction does not lock the entire adaptive hash index. Adaptive hash index partitioning is controlled by innodb\_adaptive\_hash\_index\_parts, which is set to 8 by default.

• TRX\_ADAPTIVE\_HASH\_TIMEOUT

Whether to relinquish the search latch immediately for the adaptive hash index, or reserve it across calls from MySQL. When there is no adaptive hash index contention, this value remains zero and statements reserve the latch until they finish. During times of contention, it counts down to zero, and statements release the latch immediately after each row lookup. When the adaptive hash index

search system is partitioned (controlled by innodb\_adaptive\_hash\_index\_parts), the value remains 0.

• TRX\_IS\_READ\_ONLY

A value of 1 indicates the transaction is read only.

• TRX\_AUTOCOMMIT\_NON\_LOCKING

A value of 1 indicates the transaction is a SELECT statement that does not use the FOR UPDATE or LOCK IN SHARED MODE clauses, and is executing with autocommit enabled so that the transaction contains only this one statement. When this column and TRX\_IS\_READ\_ONLY are both 1, InnoDB optimizes the transaction to reduce the overhead associated with transactions that change table data.

• TRX\_SCHEDULE\_WEIGHT

The transaction schedule weight assigned by the Contention-Aware Transaction Scheduling (CATS) algorithm to transactions waiting for a lock. The value is relative to the values of other transactions. A higher value has a greater weight. A value is computed only for transactions in a LOCK WAIT state, as reported by the TRX\_STATE column. A NULL value is reported for transactions that are not waiting for a lock. The TRX\_SCHEDULE\_WEIGHT value is different from the TRX\_WEIGHT value, which is computed by a different algorithm for a different purpose.

**Example**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_TRX\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

trx\_id: 1510

trx\_state: RUNNING

trx\_started: 2014-11-19 13:24:40

trx\_requested\_lock\_id: NULL

trx\_wait\_started: NULL

trx\_weight: 586739

trx\_mysql\_thread\_id: 2

trx\_query: DELETE FROM employees .salaries WHERE salary > 65000

trx\_operation\_state: updating or deleting

trx\_tables\_in\_use: 1

trx\_tables\_locked: 1

trx\_lock\_structs: 3003

trx\_lock\_memory\_bytes: 450768

trx\_rows\_locked: 1407513

trx\_rows\_modified: 583736

trx\_concurrency\_tickets: 0

trx\_isolation\_level: REPEATABLE READ

trx\_unique\_checks: 1

trx\_foreign\_key\_checks: 1

trx\_last\_foreign\_key\_error: NULL

trx\_adaptive\_hash\_latched: 0

trx\_adaptive\_hash\_timeout: 10000

trx\_is\_read\_only: 0

trx\_autocommit\_non\_locking: 0

trx\_schedule\_weight: NULL

**Notes**

• Use this table to help diagnose performance problems that occur during times of heavy concurrent load. Its contents are updated as described in Section 15.15.2.3, “Persistence and Consistency of InnoDB Transaction and Locking Information” .

• You must have the PROCESS privilege to query this table.

• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.4.29** **The** **INFORMATION\_SCHEMA** **INNODB\_VIRTUAL** **Table**

**WHERE** **NAME** **LIKE** **"test/t1");**

+----------+-------+----------+

| TABLE\_ID | POS | BASE\_POS |

+----------+-------+----------+

| 98 | 65538 | 0 |

| 98 | 65538 | 1 |

+----------+-------+----------+

The [INNODB\_VIRTUAL](#_bookmark147) table provides metadata about InnoDB virtual generated columns and columns upon which virtual generated columns are based.

A row appears in the [INNODB\_VIRTUAL](#_bookmark147) table for each column upon which a virtual generated column is based.

The [INNODB\_VIRTUAL](#_bookmark147) table has these columns:

• TABLE\_ID

An identifier representing the table associated with the virtual column; the same value as

INNODB\_TABLES.TABLE\_ID.

• POS

The position value of the virtual generated column. The value is large because it encodes the column sequence number and ordinal position. The formula used to calculate the value uses a bitwise operation:

((*n*th virtual generated column for the InnoDB instance + 1) << 16)

+ the ordinal position of the virtual generated column

For example, if the first virtual generated column in the InnoDB instance is the third column of the table, the formula is (0 + 1) << 16) + 2. The first virtual generated column in the InnoDB instance is always number 0. As the third column in the table, the ordinal position of the virtual generated column is 2. Ordinal positions are counted from 0.

• BASE\_POS

The ordinal position of the columns upon which a virtual generated column is based.

**Example**

mysql> **CREATE** **TABLE** **`t1`** **(**

**`a`** **int(11)** **DEFAULT** **NULL,**

**`b`** **int(11)** **DEFAULT** **NULL,**

**`c`** **int(11)** **GENERATED** **ALWAYS** **AS** **(a+b)** **VIRTUAL,**

**`h`** **varchar(10)** **DEFAULT** **NULL**

**)** **ENGINE=InnoDB** **DEFAULT** **CHARSET=utf8mb4;**

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_VIRTUAL**

**WHERE** **TABLE\_ID** **IN**

**(SELECT** **TABLE\_ID** **FROM** **INFORMATION\_SCHEMA.INNODB\_TABLES**

**Notes**

• If a constant value is assigned to a virtual generated column, as in the following table, an entry for the column does not appear in the INNODB\_VIRTUAL table. For an entry to appear, a virtual generated column must have a base column.

CREATE TABLE `t1` (

`a` int(11) DEFAULT NULL,

`b` int(11) DEFAULT NULL,

`c` int(11) GENERATED ALWAYS AS (5) VIRTUAL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

However, metadata for such a column does appear in the [INNODB\_COLUMNS](#_bookmark127) table.

• You must have the PROCESS privilege to query this table.



• Use the INFORMATION\_SCHEMA [COLUMNS](#_bookmark80) table or the SHOW COLUMNS statement to view additional information about the columns of this table, including data types and default values.

**26.5** **INFORMATION\_SCHEMA** **Thread** **Pool** **Tables**

**Note**

As of MySQL 8.0.14, the INFORMATION\_SCHEMA thread pool tables are also available as Performance Schema tables. (See Section 27.12.16, “Performance Schema Thread Pool Tables” .) The INFORMATION\_SCHEMA tables are deprecated; expect them be removed in a future version of MySQL. Applications should transition away from the old tables to the new tables. For example, if an application uses this query:

SELECT \* FROM INFORMATION\_SCHEMA.TP\_THREAD\_STATE;

The application should use this query instead:

SELECT \* FROM performance\_schema.tp\_thread\_state;

The following sections describe the INFORMATION\_SCHEMA tables associated with the thread pool plugin (see Section 5.6.3, “MySQL Enterprise Thread Pool” ). They provide information about thread pool operation:

• [TP\_THREAD\_GROUP\_STATE](#_bookmark150): Information about thread pool thread group states

• [TP\_THREAD\_GROUP\_STATS](#_bookmark151): Thread group statistics

• [TP\_THREAD\_STATE](#_bookmark152): Information about thread pool thread states

Rows in these tables represent snapshots in time. In the case of TP\_THREAD\_STATE, all rows for a thread group comprise a snapshot in time. Thus, the MySQL server holds the mutex of the thread group while producing the snapshot. But it does not hold mutexes on all thread groups at the same time, to prevent a statement against TP\_THREAD\_STATE from blocking the entire MySQL server.

The INFORMATION\_SCHEMA thread pool tables are implemented by individual plugins and the decision whether to load one can be made independently of the others (see Section 5.6.3.2, “Thread Pool Installation” ). However, the content of all the tables depends on the thread pool plugin being enabled. If a table plugin is enabled but the thread pool plugin is not, the table becomes visible and can be accessed but is empty.

**26.5.1** **INFORMATION\_SCHEMA** **Thread** **Pool** **Table** **Reference**

The following table summarizes INFORMATION\_SCHEMA thread pool tables. For greater detail, see the individual table descriptions.

**Table** **26.7** **INFORMATION\_SCHEMA** **Thread** **Pool** **Tables**

|  |  |
| --- | --- |
| **Table** **Name** | **Description** |
| [TP\_THREAD\_GROUP\_STATE](#_bookmark150) | Thread pool thread group states |
| [TP\_THREAD\_GROUP\_STATS](#_bookmark151) | Thread pool thread group statistics |
| [TP\_THREAD\_STATE](#_bookmark152) | Thread pool thread information |

**26.5.2** **The** **INFORMATION\_SCHEMA** **TP\_THREAD\_GROUP\_STATE** **Table**

**Note**

As of MySQL 8.0.14, the thread pool INFORMATION\_SCHEMA tables are also available as Performance Schema tables. (See Section 27.12.16, “Performance Schema Thread Pool Tables” .) The INFORMATION\_SCHEMA



tables are deprecated; expect them to be removed in a future version of MySQL. Applications should transition away from the old tables to the new tables. For example, if an application uses this query:

SELECT \* FROM INFORMATION\_SCHEMA.TP\_THREAD\_GROUP\_STATE;

The application should use this query instead:

SELECT \* FROM performance\_schema.tp\_thread\_group\_state;

The TP\_THREAD\_GROUP\_STATE table has one row per thread group in the thread pool. Each row provides information about the current state of a group.

For descriptions of the columns in the INFORMATION\_SCHEMA TP\_THREAD\_GROUP\_STATE table, see Section 27.12.16.1, “The tp\_thread\_group\_state Table” . The Performance Schema tp\_thread\_group\_state table has equivalent columns.

**26.5.3** **The** **INFORMATION\_SCHEMA** **TP\_THREAD\_GROUP\_STATS** **Table**

**Note**

As of MySQL 8.0.14, the thread pool INFORMATION\_SCHEMA tables are also available as Performance Schema tables. (See Section 27.12.16, “Performance Schema Thread Pool Tables” .) The INFORMATION\_SCHEMA tables are deprecated; expect them to be removed in a future version of MySQL. Applications should transition away from the old tables to the new tables. For example, if an application uses this query:

SELECT \* FROM INFORMATION\_SCHEMA.TP\_THREAD\_GROUP\_STATS;

The application should use this query instead:

SELECT \* FROM performance\_schema.tp\_thread\_group\_stats;

The TP\_THREAD\_GROUP\_STATS table reports statistics per thread group. There is one row per group.

For descriptions of the columns in the INFORMATION\_SCHEMA TP\_THREAD\_GROUP\_STATS table, see Section 27.12.16.2, “The tp\_thread\_group\_stats Table” . The Performance Schema tp\_thread\_group\_stats table has equivalent columns.

**26.5.4** **The** **INFORMATION\_SCHEMA** **TP\_THREAD\_STATE** **Table**

**Note**

As of MySQL 8.0.14, the thread pool INFORMATION\_SCHEMA tables are also available as Performance Schema tables. (See Section 27.12.16, “Performance Schema Thread Pool Tables” .) The INFORMATION\_SCHEMA tables are deprecated; expect them to be removed in a future version of MySQL. Applications should transition away from the old tables to the new tables. For example, if an application uses this query:

SELECT \* FROM INFORMATION\_SCHEMA.TP\_THREAD\_STATE;

The application should use this query instead:

SELECT \* FROM performance\_schema.tp\_thread\_state;

The TP\_THREAD\_STATE table has one row per thread created by the thread pool to handle connections.

For descriptions of the columns in the INFORMATION\_SCHEMA TP\_THREAD\_STATE table, see Section 27.12.16.3, “The tp\_thread\_state Table” . The Performance Schema tp\_thread\_state table has equivalent columns.

**26.6** **INFORMATION\_SCHEMA** **Connection-Control** **Tables**

The following sections describe the INFORMATION\_SCHEMA tables associated with the CONNECTION\_CONTROL plugin.

**26.6.1** **INFORMATION\_SCHEMA** **Connection-Control** **Table** **Reference**

The following table summarizes INFORMATION\_SCHEMA connection-control tables. For greater detail, see the individual table descriptions.

**Table** **26.8** **INFORMATION\_SCHEMA** **Connection-Control** **Tables**

|  |  |
| --- | --- |
| **Table** **Name** | **Description** |
| CONNECTION\_CONTROL\_FAILED\_LOGIN\_ATTEMP | Surrent number of consecutive failed connection attempts per account |

**26.6.2** **The** **INFORMATION\_SCHEMA**

**CONNECTION\_CONTROL\_FAILED\_LOGIN\_ATTEMPTS** **Table**

This table provides information about the current number of consecutive failed connection attempts per account (user/host combination).

[CONNECTION\_CONTROL\_FAILED\_LOGIN\_ATTEMPTS](#_bookmark155) has these columns:

• USERHOST

The user/host combination indicating an account that has failed connection attempts, in '*user\_name*'@'*host\_name*' format.

• FAILED\_ATTEMPTS

The current number of consecutive failed connection attempts for the USERHOST value. This counts all failed attempts, regardless of whether they were delayed. The number of attempts for which the server added a delay to its response is the difference between the FAILED\_ATTEMPTS value and the connection\_control\_failed\_connections\_threshold system variable value.

**Notes**

• The CONNECTION\_CONTROL\_FAILED\_LOGIN\_ATTEMPTS plugin must be activated for this table to be available, and the CONNECTION\_CONTROL plugin must be activated or the table contents are always empty. See Section 6.4.2, “The Connection-Control Plugins” .

• The table contains rows only for accounts that have had one or more consecutive failed connection attempts without a subsequent successful attempt. When an account connects successfully, its failed-connection count is reset to zero and the server removes any row corresponding to the account.

• Assigning a value to the connection\_control\_failed\_connections\_threshold system variable at runtime resets all accumulated failed-connection counters to zero, which causes the table to become empty.

**26.7** **INFORMATION\_SCHEMA** **MySQL** **Enterprise** **Firewall** **Tables**

The following sections describe the INFORMATION\_SCHEMA tables associated with MySQL Enterprise Firewall (see Section 6.4.7, “MySQL Enterprise Firewall” ). They provide views into the firewall in- memory data cache. These tables are available only if the appropriate firewall plugins are enabled.

**26.7.1** **INFORMATION\_SCHEMA** **Firewall** **Table** **Reference**

The following table summarizes INFORMATION\_SCHEMA firewall tables. For greater detail, see the individual table descriptions.

**Table** **26.9** **INFORMATION\_SCHEMA** **Firewall** **Tables**

|  |  |  |
| --- | --- | --- |
| **Table** **Name** | **Description** | **Deprecated** |
| [MYSQL\_FIREWALL\_USERS](#_bookmark158) | Firewall in-memory data for account profiles | 8.0.26 |
| [MYSQL\_FIREWALL\_WHITELIST](#_bookmark159) | Firewall in-memory data for account profile allowlists | 8.0.26 |

**26.7.2** **The** **INFORMATION\_SCHEMA** **MYSQL\_FIREWALL\_USERS** **Table**

The [MYSQL\_FIREWALL\_USERS](#_bookmark158) table provides a view into the in-memory data cache for MySQL Enterprise Firewall. It lists names and operational modes of registered firewall account profiles. It is used in conjunction with the mysql.firewall\_users system table that provides persistent storage of firewall data; see MySQL Enterprise Firewall Tables.

The [MYSQL\_FIREWALL\_USERS](#_bookmark158) table has these columns:

• USERHOST

The account profile name. Each account name has the format *user\_name*@*host\_name*.

• MODE

The current operational mode for the profile. Permitted mode values are OFF, DETECTING, PROTECTING, RECORDING, and RESET. For details about their meanings, see Firewall Concepts.

As of MySQL 8.0.26, this table is deprecated and subject to removal in a future MySQL version. See Migrating Account Profiles to Group Profiles.

**26.7.3** **The** **INFORMATION\_SCHEMA** **MYSQL\_FIREWALL\_WHITELIST** **Table**

The [MYSQL\_FIREWALL\_WHITELIST](#_bookmark159) table provides a view into the in-memory data cache for MySQL Enterprise Firewall. It lists allowlist rules of registered firewall account profiles. It is used in conjunction with the mysql.firewall\_whitelist system table that provides persistent storage of firewall data;

see MySQL Enterprise Firewall Tables.

The [MYSQL\_FIREWALL\_WHITELIST](#_bookmark159) table has these columns:

• USERHOST

The account profile name. Each account name has the format *user\_name*@*host\_name*.

• RULE

A normalized statement indicating an acceptable statement pattern for the profile. A profile allowlist is the union of its rules.

As of MySQL 8.0.26, this table is deprecated and subject to removal in a future MySQL version. See Migrating Account Profiles to Group Profiles.

**26.8** **Extensions** **to** **SHOW** **Statements**

Some extensions to SHOW statements accompany the implementation of INFORMATION\_SCHEMA:

• SHOW can be used to get information about the structure of INFORMATION\_SCHEMA itself.

• Several SHOW statements accept a WHERE clause that provides more flexibility in specifying which rows to display.

INFORMATION\_SCHEMA is an information database, so its name is included in the output from SHOW DATABASES. Similarly, SHOW TABLES can be used with INFORMATION\_SCHEMA to obtain a list of its tables:

mysql> **SHOW** **TABLES** **FROM** **INFORMATION\_SCHEMA;**

+---------------------------------------+

| Tables\_in\_INFORMATION\_SCHEMA |

+---------------------------------------+

|

COLLATIONS

COLLATION\_CHARACTER\_SET\_APPLICABILITY

COLUMNS

|

|

|

|

|

|

|

|

|

|

|

|

|

TABLES

TABLE\_CONSTRAINTS

TABLE\_PRIVILEGES

TRIGGERS

USER\_PRIVILEGES

VIEWS

+---------------------------------------+

SHOW COLUMNS and DESCRIBE can display information about the columns in individual INFORMATION\_SCHEMA tables.

COLUMN\_PRIVILEGES

ENGINES

EVENTS

FILES

KEY\_COLUMN\_USAGE

PARTITIONS

PLUGINS

PROCESSLIST REFERENTIAL\_CONSTRAINTS ROUTINES

SCHEMATA

SCHEMA\_PRIVILEGES

STATISTICS

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

CHARACTER\_SETS

|

|

|

|

|

|

|

|

|

SHOW statements that accept a LIKE clause to limit the rows displayed also permit a WHERE clause that specifies more general conditions that selected rows must satisfy:

|  |  |
| --- | --- |
| SHOW  SHOW  SHOW  SHOW  SHOW  SHOW  SHOW  SHOW  SHOW  SHOW  SHOW  SHOW  SHOW | CHARACTER SET  COLLATION  COLUMNS  DATABASES  FUNCTION STATUS  INDEX  OPEN TABLES  PROCEDURE STATUS  STATUS  TABLE STATUS  TABLES  TRIGGERS  VARIABLES |

The WHERE clause, if present, is evaluated against the column names displayed by the SHOW statement. For example, the SHOW CHARACTER SET statement produces these output columns:

mysql> **SHOW** **CHARACTER** **SET;**

+----------+-----------------------------+---------------------+--------+

| Charset | Description | Default collation | Maxlen |

+----------+-----------------------------+---------------------+--------+

big5\_chinese\_ci

dec8\_swedish\_ci

cp850\_general\_ci

hp8\_english\_ci

koi8r\_general\_ci

latin1\_swedish\_ci

latin2\_general\_ci

Big5 Traditional Chinese

DEC West European

DOS West European

HP West European

KOI8-R Relcom Russian

cp1252 West European

ISO 8859-2 Central European

big5

dec8

cp850

hp8

koi8r

latin1

latin2

2

1

1

1

1

1

1

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

...

To use a WHERE clause with SHOW CHARACTER SET, you would refer to those column names. As an example, the following statement displays information about character sets for which the default collation contains the string 'japanese':

mysql> **SHOW** **CHARACTER** **SET** **WHERE** **`Default** **collation`** **LIKE** **'%japanese%';**

+---------+---------------------------+---------------------+--------+

| Charset | Description | Default collation | Maxlen |

+---------+---------------------------+---------------------+--------+

|  |  |  |
| --- | --- | --- |
| | ujis | EUC-JP Japanese | ujis\_japanese\_ci |  | sjis | Shift-JIS Japanese | sjis\_japanese\_ci |  | cp932 | SJIS for Windows Japanese | cp932\_japanese\_ci |  | eucjpms | UJIS for Windows Japanese | eucjpms\_japanese\_ci | | 3  2  2  3 | |  |  |  | |

+---------+---------------------------+---------------------+--------+

This statement displays the multibyte character sets:

mysql> **SHOW** **CHARACTER** **SET** **WHERE** **Maxlen** **>** **1;**

+---------+---------------------------------+---------------------+--------+

| Charset | Description | Default collation | Maxlen |

+---------+---------------------------------+---------------------+--------+

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

+---------+---------------------------------+---------------------+--------+

Big5 Traditional Chinese

SJIS for Windows Japanese

UJIS for Windows Japanese

EUC-KR Korean

China National Standard GB18030

GB2312 Simplified Chinese

GBK Simplified Chinese

Shift-JIS Japanese

UCS-2 Unicode

EUC-JP Japanese

UTF-16 Unicode

UTF-16LE Unicode

UTF-32 Unicode

UTF-8 Unicode

UTF-8 Unicode

big5\_chinese\_ci

cp932\_japanese\_ci

eucjpms\_japanese\_ci

euckr\_korean\_ci

gb18030\_chinese\_ci

gb2312\_chinese\_ci

gbk\_chinese\_ci

sjis\_japanese\_ci

ucs2\_general\_ci

ujis\_japanese\_ci

utf16\_general\_ci

utf16le\_general\_ci

utf32\_general\_ci

utf8mb3\_general\_ci

utf8mb4\_0900\_ai\_ci

big5

cp932

eucjpms

euckr

gb18030

gb2312

gbk

sjis

ucs2

ujis

utf16

utf16le

utf32

utf8mb3

utf8mb4

2

2

3

2

4

2

2

2

2

3

4

4

4

3

4

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

Chapter 27 MySQL Performance Schema

**Table** **of** **Contents**

[27.1 Performance Schema Quick Start](#_bookmark173) [4955](#_bookmark173)

[27.2 Performance Schema Build Configuration](#_bookmark174) [4961](#_bookmark174)

[27.3 Performance Schema Startup Configuration](#_bookmark175) [4961](#_bookmark175)

[27.4 Performance Schema Runtime Configuration](#_bookmark176) [4963](#_bookmark176)

[27.4.1 Performance Schema Event Timing](#_bookmark177) [4964](#_bookmark177)

[27.4.2 Performance Schema Event Filtering](#_bookmark178) [4966](#_bookmark178)

[27.4.3 Event Pre-Filtering](#_bookmark179) [4968](#_bookmark179)

[27.4.4 Pre-Filtering Instrument](#_bookmark180)by [4968](#_bookmark180)

[27.4.5 Pre-Filtering Object](#_bookmark181)by [4970](#_bookmark181)

[27.4.6 Pre-Filtering Thread](#_bookmark182)by [4971](#_bookmark182)

27.4.7 Pre-Filtering by Consumer 4973

27.4.8 Example Consumer Configurations 4976

27.4.9 Naming Instruments or Consumers for Filtering Operations 4981

27.4.10 Determining What Is Instrumented 4981

27.5 Performance Schema Queries 4982

27.6 Performance Schema Instrument Naming Conventions 4982

27.7 Performance Schema Status Monitoring 4986

27.8 Performance Schema Atom and Molecule Events 4989

27.9 Performance Schema Tables for Current and Historical Events 4989

27.10 Performance Schema Statement Digests and Sampling 4991

27.11 Performance Schema General Table Characteristics 4995

27.12 Performance Schema Table Descriptions 4996

27.12.1 Performance Schema Table Reference 4996

27.12.2 Performance Schema Setup Tables 5001

27.12.3 Performance Schema Instance Tables 5010

27.12.4 Performance Schema Wait Event Tables 5015

27.12.5 Performance Schema Stage Event Tables 5020

27.12.6 Performance Schema Statement Event Tables 5026

27.12.7 Performance Schema Transaction Tables 5037

27.12.8 Performance Schema Connection Tables 5044

27.12.9 Performance Schema Connection Attribute Tables 5048

27.12.10 Performance Schema User-Defined Variable Tables 5053

27.12.11 Performance Schema Replication Tables 5053

27.12.12 Performance Schema NDB Cluster Tables 5076

27.12.13 Performance Schema Lock Tables 5079

27.12.14 Performance Schema System Variable Tables 5088

27.12.15 Performance Schema Status Variable Tables 5093

27.12.16 Performance Schema Thread Pool Tables 5094

27.12.17 Performance Schema Firewall Tables 5099

27.12.18 Performance Schema Keyring Tables 5101

27.12.19 Performance Schema Clone Tables 5102

27.12.20 Performance Schema Summary Tables 5105

27.12.21 Performance Schema Miscellaneous Tables 5133

27.13 Performance Schema Option and Variable Reference 5151

27.14 Performance Schema Command Options 5155

27.15 Performance Schema System Variables 5156

27.16 Performance Schema Status Variables 5175

27.17 The Performance Schema Memory-Allocation Model 5178

27.18 Performance Schema and Plugins 5179

27.19 Using the Performance Schema to Diagnose Problems 5179

27.19.1 Query Profiling Using Performance Schema 5180

27.19.2 Obtaining Parent Event Information 5182

27.20 Restrictions on Performance Schema . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5184

The MySQL Performance Schema is a feature for monitoring MySQL Server execution at a low level. The Performance Schema has these characteristics:

• The Performance Schema provides a way to inspect internal execution of the server at runtime. It is implemented using the [PERFORMANCE\_SCHEMA](#_bookmark171) storage engine and the performance\_schema database. The Performance Schema focuses primarily on performance data. This differs from INFORMATION\_SCHEMA, which serves for inspection of metadata.

• The Performance Schema monitors server events. An “event” is anything the server does that takes time and has been instrumented so that timing information can be collected. In general, an event could be a function call, a wait for the operating system, a stage of an SQL statement execution such as parsing or sorting, or an entire statement or group of statements. Event collection provides access to information about synchronization calls (such as for mutexes) file and table I/O, table locks, and so forth for the server and for several storage engines.

• Performance Schema events are distinct from events written to the server's binary log (which describe data modifications) and Event Scheduler events (which are a type of stored program).

• Performance Schema events are specific to a given instance of the MySQL Server. Performance Schema tables are considered local to the server, and changes to them are not replicated or written to the binary log.

• Current events are available, as well as event histories and summaries. This enables you to determine how many times instrumented activities were performed and how much time they took. Event information is available to show the activities of specific threads, or activity associated with particular objects such as a mutex or file.

• The [PERFORMANCE\_SCHEMA](#_bookmark171) storage engine collects event data using “instrumentation points” in server source code.

• Collected events are stored in tables in the performance\_schema database. These tables can be queried using SELECT statements like other tables.

• Performance Schema configuration can be modified dynamically by updating tables in the performance\_schema database through SQL statements. Configuration changes affect data collection immediately.

• Tables in the Performance Schema are in-memory tables that use no persistent on-disk storage. The contents are repopulated beginning at server startup and discarded at server shutdown.

• Monitoring is available on all platforms supported by MySQL.

Some limitations might apply: The types of timers might vary per platform. Instruments that apply to storage engines might not be implemented for all storage engines. Instrumentation of each third- party engine is the responsibility of the engine maintainer. See also Section 27.20, “Restrictions on

Performance Schema” .

• Data collection is implemented by modifying the server source code to add instrumentation. There are no separate threads associated with the Performance Schema, unlike other features such as replication or the Event Scheduler.

The Performance Schema is intended to provide access to useful information about server execution while having minimal impact on server performance. The implementation follows these design goals:

• Activating the Performance Schema causes no changes in server behavior. For example, it does not cause thread scheduling to change, and it does not cause query execution plans (as shown by EXPLAIN) to change.

• Server monitoring occurs continuously and unobtrusively with very little overhead. Activating the Performance Schema does not make the server unusable.



• The parser is unchanged. There are no new keywords or statements.

• Execution of server code proceeds normally even if the Performance Schema fails internally.

• When there is a choice between performing processing during event collection initially or during event retrieval later, priority is given to making collection faster. This is because collection is ongoing whereas retrieval is on demand and might never happen at all.

• Most Performance Schema tables have indexes, which gives the optimizer access to execution plans other than full table scans. For more information, see Section 8.2.4, “Optimizing Performance Schema Queries” .

• It is easy to add new instrumentation points.

• Instrumentation is versioned. If the instrumentation implementation changes, previously instrumented code continues to work. This benefits developers of third-party plugins because it is not necessary to upgrade each plugin to stay synchronized with the latest Performance Schema changes.

**Note**

The MySQL sys schema is a set of objects that provides convenient access to data collected by the Performance Schema. The sys schema is installed by default. For usage instructions, see Chapter 28, *MySQL* *sys* *Schema*.

**27.1** **Performance** **Schema** **Quick** **Start**

This section briefly introduces the Performance Schema with examples that show how to use it. For additional examples, see Section 27.19, “Using the Performance Schema to Diagnose Problems” .

The Performance Schema is enabled by default. To enable or disable it explicitly, start the server with the performance\_schema variable set to an appropriate value. For example, use these lines in the server my.cnf file:

[mysqld]

performance\_schema=ON

When the server starts, it sees performance\_schema and attempts to initialize the Performance Schema. To verify successful initialization, use this statement:

mysql> **SHOW** **VARIABLES** **LIKE** **'performance\_schema';**

+--------------------+-------+

| Variable\_name | Value |

+--------------------+-------+

| performance\_schema | ON |

+--------------------+-------+

A value of ON means that the Performance Schema initialized successfully and is ready for use. A value of OFF means that some error occurred. Check the server error log for information about what went wrong.

The Performance Schema is implemented as a storage engine, so you can see it listed in the output from the Information Schema [ENGINES](#_bookmark85) table or the SHOW ENGINES statement:

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.ENGINES**

**WHERE** **ENGINE='PERFORMANCE\_SCHEMA'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ENGINE: PERFORMANCE\_SCHEMA

SUPPORT: YES

COMMENT: Performance Schema

TRANSACTIONS: NO

XA: NO

SAVEPOINTS: NO

mysql> **SHOW** **ENGINES\G**

...

Engine: PERFORMANCE\_SCHEMA

Support: YES

Comment: Performance Schema

Transactions: NO

XA: NO

Savepoints: NO

...

The [PERFORMANCE\_SCHEMA](#_bookmark171) storage engine operates on tables in the performance\_schema database. You can make performance\_schema the default database so that references to its tables need not be qualified with the database name:

mysql> **USE** **performance\_schema;**

Performance Schema tables are stored in the performance\_schema database. Information about the structure of this database and its tables can be obtained, as for any other database, by selecting from the INFORMATION\_SCHEMA database or by using SHOW statements. For example, use either of these statements to see what Performance Schema tables exist:

mysql> **SELECT** **TABLE\_NAME** **FROM** **INFORMATION\_SCHEMA.TABLES**

**WHERE** **TABLE\_SCHEMA** **=** **'performance\_schema';**

+------------------------------------------------------+

| TABLE\_NAME

|

+------------------------------------------------------+

| accounts

| cond\_instances

...

| events\_stages\_current

| events\_stages\_history

| events\_stages\_history\_long

| events\_stages\_summary\_by\_account\_by\_event\_name

| events\_stages\_summary\_by\_host\_by\_event\_name

| events\_stages\_summary\_by\_thread\_by\_event\_name

| events\_stages\_summary\_by\_user\_by\_event\_name

| events\_stages\_summary\_global\_by\_event\_name

| events\_statements\_current

| events\_statements\_history

| events\_statements\_history\_long

...

| file\_instances

| file\_summary\_by\_event\_name

| file\_summary\_by\_instance

| host\_cache

| hosts

| memory\_summary\_by\_account\_by\_event\_name

| memory\_summary\_by\_host\_by\_event\_name

| memory\_summary\_by\_thread\_by\_event\_name

| memory\_summary\_by\_user\_by\_event\_name

| memory\_summary\_global\_by\_event\_name

| metadata\_locks

| mutex\_instances

| objects\_summary\_global\_by\_type

| performance\_timers

| replication\_connection\_configuration

| replication\_connection\_status

| replication\_applier\_configuration

| replication\_applier\_status

| replication\_applier\_status\_by\_coordinator

| replication\_applier\_status\_by\_worker

| rwlock\_instances

| session\_account\_connect\_attrs

| session\_connect\_attrs

| setup\_actors

| setup\_consumers

| setup\_instruments

| setup\_objects

| socket\_instances

| socket\_summary\_by\_event\_name

| socket\_summary\_by\_instance

| table\_handles

| table\_io\_waits\_summary\_by\_index\_usage

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|  |  |
| --- | --- |
| | table\_io\_waits\_summary\_by\_table  | table\_lock\_waits\_summary\_by\_table  | threads  | users | |  |  |  | |

+------------------------------------------------------+

mysql> **SHOW** **TABLES** **FROM** **performance\_schema;**

+------------------------------------------------------+

| Tables\_in\_performance\_schema |

+------------------------------------------------------+

|  |  |
| --- | --- |
| | accounts  | cond\_instances  | events\_stages\_current  | events\_stages\_history  | events\_stages\_history\_long  ... | |  |  |  |  | |

The number of Performance Schema tables increases over time as implementation of additional instrumentation proceeds.

The name of the performance\_schema database is lowercase, as are the names of tables within it. Queries should specify the names in lowercase.

To see the structure of individual tables, use SHOW CREATE TABLE:

mysql> **SHOW** **CREATE** **TABLE** **performance\_schema** **.setup\_consumers\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Table: setup\_consumers

Create Table: CREATE TABLE `setup\_consumers` (

`NAME` varchar(64) NOT NULL,

`ENABLED` enum('YES','NO') NOT NULL,

PRIMARY KEY (`NAME`)

) ENGINE=PERFORMANCE\_SCHEMA DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_0900\_ai\_ci

Table structure is also available by selecting from tables such as [INFORMATION\_SCHEMA.COLUMNS](#_bookmark80) or by using statements such as SHOW COLUMNS.

Tables in the performance\_schema database can be grouped according to the type of information in them: Current events, event histories and summaries, object instances, and setup (configuration) information. The following examples illustrate a few uses for these tables. For detailed information about the tables in each group, see Section 27.12, “Performance Schema Table Descriptions” .

Initially, not all instruments and consumers are enabled, so the performance schema does not collect all events. To turn all of these on and enable event timing, execute two statements (the row counts may differ depending on MySQL version):

mysql> **UPDATE** **performance\_schema.setup\_instruments**

**SET** **ENABLED** **=** **'YES',** **TIMED** **=** **'YES';**

Query OK, 560 rows affected (0.04 sec)

mysql> **UPDATE** **performance\_schema** **.setup\_consumers**

**SET** **ENABLED** **=** **'YES';**

Query OK, 10 rows affected (0.00 sec)

To see what the server is doing at the moment, examine the events\_waits\_current table. It contains one row per thread showing each thread's most recent monitored event:

mysql> **SELECT** **\***

**FROM** **performance\_schema** **.events\_waits\_current\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

THREAD\_ID: 0

EVENT\_ID: 5523

END\_EVENT\_ID: 5523

EVENT\_NAME: wait/synch/mutex/mysys/THR\_LOCK::mutex

SOURCE: thr\_lock.c:525

TIMER\_START: 201660494489586

TIMER\_END: 201660494576112

TIMER\_WAIT: 86526

SPINS: NULL

OBJECT\_SCHEMA: NULL

OBJECT\_NAME: NULL

INDEX\_NAME: NULL

OBJECT\_TYPE: NULL

OBJECT\_INSTANCE\_BEGIN: 142270668

NESTING\_EVENT\_ID: NULL

NESTING\_EVENT\_TYPE: NULL

OPERATION: lock

NUMBER\_OF\_BYTES: NULL

FLAGS: 0

...

This event indicates that thread 0 was waiting for 86,526 picoseconds to acquire a lock on THR\_LOCK::mutex, a mutex in the mysys subsystem. The first few columns provide the following information:

• The ID columns indicate which thread the event comes from and the event number.

• EVENT\_NAME indicates what was instrumented and SOURCE indicates which source file contains the instrumented code.

• The timer columns show when the event started and stopped and how long it took. If an event is still in progress, the TIMER\_END and TIMER\_WAIT values are NULL. Timer values are approximate and expressed in picoseconds. For information about timers and event time collection, see [Section 27.4.1, “Performance Schema Event Timing”](#_bookmark177) .

The history tables contain the same kind of rows as the current-events table but have more rows and show what the server has been doing “recently” rather than “currently.” The events\_waits\_history and events\_waits\_history\_long tables contain the most recent 10 events per thread and most recent 10,000 events, respectively. For example, to see information for recent events produced by thread 13, do this:

mysql> **SELECT** **EVENT\_ID,** **EVENT\_NAME,** **TIMER\_WAIT**

**FROM** **performance\_schema.events\_waits\_history**

**WHERE** **THREAD\_ID** **=** **13**

**ORDER** **BY** **EVENT\_ID;**

+----------+-----------------------------------------+------------+

|  |  |
| --- | --- |
| | EVENT\_ID | EVENT\_NAME | | TIMER\_WAIT | |

+----------+-----------------------------------------+------------+

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| | | 86 | | | wait/synch/mutex/mysys/THR\_LOCK::mutex | | | 686322 | | |
| | | 87 | | | wait/synch/mutex/mysys/THR\_LOCK\_malloc | | | 320535 | | |
| | | 88 | | | wait/synch/mutex/mysys/THR\_LOCK\_malloc | | | 339390 | | |
| | | 89 | | | wait/synch/mutex/mysys/THR\_LOCK\_malloc | | | 377100 | | |
| | | 90 | | | wait/synch/mutex/sql/LOCK\_plugin | | | 614673 | | |
| | | 91 | | | wait/synch/mutex/sql/LOCK\_open | | | 659925 | | |
| | | 92 | | | wait/synch/mutex/sql/THD::LOCK\_thd\_data | | | 494001 | | |
| | | 93 | | | wait/synch/mutex/mysys/THR\_LOCK\_malloc | | | 222489 | | |
| | | 94 | | | wait/synch/mutex/mysys/THR\_LOCK\_malloc | | | 214947 | | |
| | | 95 | | | wait/synch/mutex/mysys/LOCK\_alarm | | | 312993 | | |

+----------+-----------------------------------------+------------+

As new events are added to a history table, older events are discarded if the table is full.

Summary tables provide aggregated information for all events over time. The tables in this group summarize event data in different ways. To see which instruments have been executed the most times or have taken the most wait time, sort the events\_waits\_summary\_global\_by\_event\_name table on the COUNT\_STAR or SUM\_TIMER\_WAIT column, which correspond to a COUNT(\*) or SUM(TIMER\_WAIT) value, respectively, calculated over all events:

mysql> **SELECT** **EVENT\_NAME,** **COUNT\_STAR**

**FROM** **performance\_schema.events\_waits\_summary\_global\_by\_event\_name**

**ORDER** **BY** **COUNT\_STAR** **DESC** **LIMIT** **10;**

+---------------------------------------------------+------------+

|  |  |
| --- | --- |
| | EVENT\_NAME | | COUNT\_STAR | |

+---------------------------------------------------+------------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | wait/synch/mutex/mysys/THR\_LOCK\_malloc wait/io/file/sql/FRM | |  | | 6419  452 | |  | |



|  |  |  |  |
| --- | --- | --- | --- |
| | | wait/synch/mutex/sql/LOCK\_plugin | | 337 | | |
| | | wait/synch/mutex/mysys/THR\_LOCK\_open | | 187 | | |
| | | wait/synch/mutex/mysys/LOCK\_alarm | | 147 | | |
| | | wait/synch/mutex/sql/THD::LOCK\_thd\_data | | 115 | | |
| | | wait/io/file/myisam/kfile | | 102 | | |
| | | wait/synch/mutex/sql/LOCK\_global\_system\_variables | | 89 | | |
| | | wait/synch/mutex/mysys/THR\_LOCK::mutex | | 89 | | |
| | | wait/synch/mutex/sql/LOCK\_open | | 88 | | |

+---------------------------------------------------+------------+

mysql> **SELECT** **EVENT\_NAME,** **SUM\_TIMER\_WAIT**

**FROM** **performance\_schema.events\_waits\_summary\_global\_by\_event\_name**

**ORDER** **BY** **SUM\_TIMER\_WAIT** **DESC** **LIMIT** **10;**

+----------------------------------------+----------------+

| EVENT\_NAME | SUM\_TIMER\_WAIT |

+----------------------------------------+----------------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| | | wait/io/file/sql/MYSQL\_LOG | | | 1599816582 | | |
| | | wait/synch/mutex/mysys/THR\_LOCK\_malloc | | | 1530083250 | | |
| | | wait/io/file/sql/binlog\_index | | | 1385291934 | | |
| | | wait/io/file/sql/FRM | | | 1292823243 | | |
| | | wait/io/file/myisam/kfile | | | 411193611 | | |
| | | wait/io/file/myisam/dfile | | | 322401645 | | |
| | | wait/synch/mutex/mysys/LOCK\_alarm | | | 145126935 | | |
| | | wait/io/file/sql/casetest | | | 104324715 | | |
| | | wait/synch/mutex/sql/LOCK\_plugin | | | 86027823 | | |
| | | wait/io/file/sql/pid | | | 72591750 | | |

+----------------------------------------+----------------+

These results show that the THR\_LOCK\_malloc mutex is “hot,” both in terms of how often it is used and amount of time that threads wait attempting to acquire it.

**Note**

The THR\_LOCK\_malloc mutex is used only in debug builds. In production builds it is not hot because it is nonexistent.

Instance tables document what types of objects are instrumented. An instrumented object, when used by the server, produces an event. These tables provide event names and explanatory notes or status information. For example, the file\_instances table lists instances of instruments for file I/O operations and their associated files:

mysql> **SELECT** **\***

**FROM** **performance\_schema** **.file\_instances\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FILE\_NAME: /opt/mysql-log/60500/binlog .000007

EVENT\_NAME: wait/io/file/sql/binlog

OPEN\_COUNT: 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FILE\_NAME: /opt/mysql/60500/data/mysql/tables\_priv .MYI

EVENT\_NAME: wait/io/file/myisam/kfile

OPEN\_COUNT: 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FILE\_NAME: /opt/mysql/60500/data/mysql/columns\_priv .MYI

EVENT\_NAME: wait/io/file/myisam/kfile

OPEN\_COUNT: 1

...

Setup tables are used to configure and display monitoring characteristics. For example, setup\_instruments lists the set of instruments for which events can be collected and shows which of them are enabled:

mysql> **SELECT** **NAME,** **ENABLED,** **TIMED**

**FROM** **performance\_schema** **.setup\_instruments;**

+---------------------------------------------------+---------+-------+

|  |  |
| --- | --- |
| | NAME | | ENABLED | TIMED | |

+---------------------------------------------------+---------+-------+

...

|

|

| stage/sql/end

| stage/sql/executing

NO

NO

NO

NO

|

|

|

|

| stage/sql/init

| stage/sql/insert

...

| statement/sql/load

| statement/sql/grant

| statement/sql/check

| statement/sql/flush

...

| wait/synch/mutex/sql/LOCK\_global\_read\_lock

| wait/synch/mutex/sql/LOCK\_global\_system\_variables

| wait/synch/mutex/sql/LOCK\_lock\_db

| wait/synch/mutex/sql/LOCK\_manager

...

| wait/synch/rwlock/sql/LOCK\_grant

| wait/synch/rwlock/sql/LOGGER::LOCK\_logger

| wait/synch/rwlock/sql/LOCK\_sys\_init\_connect

| wait/synch/rwlock/sql/LOCK\_sys\_init\_slave

...

| wait/io/file/sql/binlog

| wait/io/file/sql/binlog\_index

| wait/io/file/sql/casetest

| wait/io/file/sql/dbopt

...

To understand how to interpret instrument names, see Section 27.6, Naming Conventions” .

“Performance Schema Instrument

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

YES

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

NO

NO

NO

NO

To control whether events are collected for an instrument, set its ENABLED value to YES or NO. For example:

mysql> **UPDATE** **performance\_schema.setup\_instruments**

**SET** **ENABLED** **=** **'NO'**

**WHERE** **NAME** **=** **'wait/synch/mutex/sql/LOCK\_mysql\_create\_db';**

The Performance Schema uses collected events to update tables in the performance\_schema database, which act as “consumers” of event information. The setup\_consumers table lists the

available consumers and which are enabled:

mysql> **SELECT** **\*** **FROM** **performance\_schema** **.setup\_consumers;**

+----------------------------------+---------+

| ENABLED |

+----------------------------------+---------+

events\_stages\_current

events\_stages\_history

events\_stages\_history\_long

events\_statements\_cpu

events\_statements\_current

events\_statements\_history

events\_statements\_history\_long

events\_transactions\_current

events\_transactions\_history

events\_transactions\_history\_long

events\_waits\_current

events\_waits\_history

events\_waits\_history\_long

global\_instrumentation

thread\_instrumentation

statements\_digest

+----------------------------------+---------+

To control whether the Performance Schema maintains a consumer as a destination for event information, set its ENABLED value.

NO

NO

NO

NO

YES

YES

NO

YES

YES

NO

NO

NO

NO

YES

YES

YES

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

| NAME

For more information about the setup tables and how to use them to control event collection, see [Section 27.4.2, “Performance Schema Event Filtering”](#_bookmark178) .

There are some miscellaneous tables that do not fall into any of the previous groups. For example, performance\_timers lists the available event timers and their characteristics. For information about timers, see [Section 27.4.1, “Performance Schema Event Timing”](#_bookmark177) .

**-DDISABLE\_PSI\_STATEMENT=1**

**27.2** **Performance** **Schema** **Build** **Configuration**

The Performance Schema is mandatory and always compiled in. It is possible to exclude certain parts of the Performance Schema instrumentation. For example, to exclude stage and statement instrumentation, do this:

$> **cmake** **.** **\**

**-DDISABLE\_PSI\_STAGE=1** **\**

For more information, see the descriptions of the DISABLE\_PSI\_*XXX* CMake options in Section 2.8.7, “MySQL Source-Configuration Options” .

If you install MySQL over a previous installation that was configured without the Performance Schema (or with an older version of the Performance Schema that has missing or out-of-date tables). One indication of this issue is the presence of messages such as the following in the error log:

[ERROR] Native table 'performance\_schema'. 'events\_waits\_history'

has the wrong structure

[ERROR] Native table 'performance\_schema' . 'events\_waits\_history\_long'

has the wrong structure

...

To correct that problem, perform the MySQL upgrade procedure. See Section 2.10, “Upgrading MySQL” .

Because the Performance Schema is configured into the server at build time, a row for [PERFORMANCE\_SCHEMA](#_bookmark171) appears in the output from SHOW ENGINES. This means that the Performance Schema is available, not that it is enabled. To enable it, you must do so at server startup, as described in the next section.

**27.3** **Performance** **Schema** **Startup** **Configuration**

To use the MySQL Performance Schema, it must be enabled at server startup to enable event collection to occur.

The Performance Schema is enabled by default. To enable or disable it explicitly, start the server with the performance\_schema variable set to an appropriate value. For example, use these lines in the server my.cnf file:

[mysqld]

performance\_schema=ON

If the server is unable to allocate any internal buffer during Performance Schema initialization, the Performance Schema disables itself and sets performance\_schema to OFF, and the server runs without instrumentation.

The Performance Schema also permits instrument and consumer configuration at server startup. To control an instrument at server startup, use an option of this form:

--performance-schema-instrument='*instrument\_name*=*value*'

Here, *instrument\_name* is an instrument name such as wait/synch/mutex/sql/LOCK\_open, and *value* is one of these values:

• OFF, FALSE, or 0: Disable the instrument

• ON, TRUE, or 1: Enable and time the instrument

• COUNTED: Enable and count (rather than time) the instrument

Each --performance-schema-instrument option can specify only one instrument name, but multiple instances of the option can be given to configure multiple instruments. In addition, patterns

are permitted in instrument names to configure instruments that match the pattern. To configure all condition synchronization instruments as enabled and counted, use this option:

--performance-schema-instrument='wait/synch/cond/%=COUNTED'

To disable all instruments, use this option:

--performance-schema-instrument='%=OFF'

Exception: The memory/performance\_schema/% instruments are built in and cannot be disabled at startup.

Longer instrument name strings take precedence over shorter pattern names, regardless of order. For information about specifying patterns to select instruments, see Section 27.4.9, “Naming Instruments or Consumers for Filtering Operations” .

An unrecognized instrument name is ignored. It is possible that a plugin installed later may create the instrument, at which time the name is recognized and configured.

To control a consumer at server startup, use an option of this form:

--performance-schema-consumer-*consumer\_name*=*value*

Here, *consumer\_name* is a consumer name such as events\_waits\_history, and *value* is one of these values:

• OFF, FALSE, or 0: Do not collect events for the consumer

• ON, TRUE, or 1: Collect events for the consumer

For example, to enable the events\_waits\_history consumer, use this option:

--performance-schema-consumer-events-waits-history=ON

The permitted consumer names can be found by examining the setup\_consumers table. Patterns are not permitted. Consumer names in the setup\_consumers table use underscores, but for consumers set at startup, dashes and underscores within the name are equivalent.

The Performance Schema includes several system variables that provide configuration information:

mysql> **SHOW** **VARIABLES** **LIKE** **'perf%';**

+--------------------------------------------------------+---------+

| Variable\_name | Value |

+--------------------------------------------------------+---------+

| performance\_schema | ON |

| performance\_schema\_accounts\_size | 100 |

| performance\_schema\_digests\_size | 200 |

| performance\_schema\_events\_stages\_history\_long\_size | 10000 |

| performance\_schema\_events\_stages\_history\_size | 10 |

| performance\_schema\_events\_statements\_history\_long\_size | 10000 |

| performance\_schema\_events\_statements\_history\_size | 10 |

| performance\_schema\_events\_waits\_history\_long\_size | 10000 |

| performance\_schema\_events\_waits\_history\_size | 10 |

| performance\_schema\_hosts\_size | 100 |

| performance\_schema\_max\_cond\_classes | 80 |

| performance\_schema\_max\_cond\_instances | 1000 |

...

The performance\_schema variable is ON or OFF to indicate whether the Performance Schema is

enabled or disabled. The other variables indicate table sizes (number of rows) or memory allocation values.

**Note**

With the Performance Schema enabled, the number of Performance Schema instances affects the server memory footprint, perhaps to a large extent. The Performance Schema autoscales many parameters to use memory only as

**AND** **TABLE\_NAME** **LIKE** **'setup%';**

+-------------------+

| TABLE\_NAME |

+-------------------+

| setup\_actors |

| setup\_consumers |

| setup\_instruments |

| setup\_objects |

| setup\_threads |

+-------------------+

required; see Section 27.17, “The Performance Schema Memory-Allocation

Model” .

To change the value of Performance Schema system variables, set them at server startup. For example, put the following lines in a my.cnf file to change the sizes of the history tables for wait events:

[mysqld]

performance\_schema

performance\_schema\_events\_waits\_history\_size=20

performance\_schema\_events\_waits\_history\_long\_size=15000

The Performance Schema automatically sizes the values of several of its parameters at server startup if they are not set explicitly. For example, it sizes the parameters that control the sizes of the events waits tables this way. The Performance Schema allocates memory incrementally, scaling its memory use to actual server load, instead of allocating all the memory it needs during server startup. Consequently, many sizing parameters need not be set at all. To see which parameters are autosized or autoscaled, use mysqld --verbose --help and examine the option descriptions, or see Section 27.15, “Performance Schema System Variables” .

For each autosized parameter that is not set at server startup, the Performance Schema determines how to set its value based on the value of the following system values, which are considered as “hints” about how you have configured your MySQL server:

max\_connections

open\_files\_limit

table\_definition\_cache

table\_open\_cache

To override autosizing or autoscaling for a given parameter, set it to a value other than − 1 at startup. In this case, the Performance Schema assigns it the specified value.

At runtime, SHOW VARIABLES displays the actual values that autosized parameters were set to. Autoscaled parameters display with a value of − 1.

If the Performance Schema is disabled, its autosized and autoscaled parameters remain set to − 1 and SHOW VARIABLES displays − 1.

**27.4** **Performance** **Schema** **Runtime** **Configuration**

Specific Performance Schema features can be enabled at runtime to control which types of event collection occur.

Performance Schema setup tables contain information about monitoring configuration:

mysql> **SELECT** **TABLE\_NAME** **FROM** **INFORMATION\_SCHEMA.TABLES**

**WHERE** **TABLE\_SCHEMA** **=** **'performance\_schema'**

You can examine the contents of these tables to obtain information about Performance Schema monitoring characteristics. If you have the UPDATE privilege, you can change Performance Schema operation by modifying setup tables to affect how monitoring occurs. For additional details about these tables, see Section 27.12.2, “Performance Schema Setup Tables” .

The setup\_instruments and setup\_consumers tables list the instruments for which events can be collected and the types of consumers for which event information actually is collected, respectively.

Other setup tables enable further modification of the monitoring configuration. [Section 27.4.2,](#_bookmark178) [“Performance Schema Event Filtering”](#_bookmark178) , discusses how you can modify these tables to affect event collection.

If there are Performance Schema configuration changes that must be made at runtime using SQL statements and you would like these changes to take effect each time the server starts, put the statements in a file and start the server with the init\_file system variable set to name the file. This strategy can also be useful if you have multiple monitoring configurations, each tailored to produce a different kind of monitoring, such as casual server health monitoring, incident investigation, application behavior troubleshooting, and so forth. Put the statements for each monitoring configuration into their own file and specify the appropriate file as the init\_file value when you start the server.

**27.4.1** **Performance** **Schema** **Event** **Timing**

Events are collected by means of instrumentation added to the server source code. Instruments time events, which is how the Performance Schema provides an idea of how long events take. It is also possible to configure instruments not to collect timing information. This section discusses the available timers and their characteristics, and how timing values are represented in events.

**Performance** **Schema** **Timers**

Performance Schema timers vary in precision and amount of overhead. To see what timers are available and their characteristics, check the performance\_timers table:

mysql> **SELECT** **\*** **FROM** **performance\_schema** **.performance\_timers;**

+-------------+-----------------+------------------+----------------+

| TIMER\_NAME | TIMER\_FREQUENCY | TIMER\_RESOLUTION | TIMER\_OVERHEAD |

+-------------+-----------------+------------------+----------------+

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | CYCLE | | | 2389029850 | | | 1 | | | 72 | | |
| | | NANOSECOND | | | 1000000000 | | | 1 | | | 112 | | |
| | | MICROSECOND | | | 1000000 | | | 1 | | | 136 | | |
| | | MILLISECOND | | | 1036 | | | 1 | | | 168 | | |
| | | THREAD\_CPU | | | 339101694 | | | 1 | | | 798 | | |

+-------------+-----------------+------------------+----------------+

If the values associated with a given timer name are NULL, that timer is not supported on your platform. The columns have these meanings:

• The TIMER\_NAME column shows the names of the available timers. CYCLE refers to the timer that is based on the CPU (processor) cycle counter.

• TIMER\_FREQUENCY indicates the number of timer units per second. For a cycle timer, the frequency is generally related to the CPU speed. The value shown was obtained on a system with a 2.4GHz processor. The other timers are based on fixed fractions of seconds.

• TIMER\_RESOLUTION indicates the number of timer units by which timer values increase at a time. If a timer has a resolution of 10, its value increases by 10 each time.

• TIMER\_OVERHEAD is the minimal number of cycles of overhead to obtain one timing with the given timer. The overhead per event is twice the value displayed because the timer is invoked at the beginning and end of the event.

The Performance Schema assigns timers as follows:

• The wait timer uses CYCLE.

• The idle, stage, statement, and transaction timers use NANOSECOND on platforms where the NANOSECOND timer is available, MICROSECOND otherwise.

At server startup, the Performance Schema verifies that assumptions made at build time about timer assignments are correct, and displays a warning if a timer is not available.

To time wait events, the most important criterion is to reduce overhead, at the possible expense of the timer accuracy, so using the CYCLE timer is the best.

The time a statement (or stage) takes to execute is in general orders of magnitude larger than the time it takes to execute a single wait. To time statements, the most important criterion is to have an accurate measure, which is not affected by changes in processor frequency, so using a timer which is not based on cycles is the best. The default timer for statements is NANOSECOND. The extra “overhead” compared to the CYCLE timer is not significant, because the overhead caused by calling a timer twice (once when the statement starts, once when it ends) is orders of magnitude less compared to the CPU time used to execute the statement itself. Using the CYCLE timer has no benefit here, only drawbacks.

The precision offered by the cycle counter depends on processor speed. If the processor runs at 1 GHz (one billion cycles/second) or higher, the cycle counter delivers sub-nanosecond precision. Using the cycle counter is much cheaper than getting the actual time of day. For example, the standard gettimeofday() function can take hundreds of cycles, which is an unacceptable overhead for data gathering that may occur thousands or millions of times per second.

Cycle counters also have disadvantages:

• End users expect to see timings in wall-clock units, such as fractions of a second. Converting from cycles to fractions of seconds can be expensive. For this reason, the conversion is a quick and fairly rough multiplication operation.

• Processor cycle rate might change, such as when a laptop goes into power-saving mode or when a CPU slows down to reduce heat generation. If a processor's cycle rate fluctuates, conversion from cycles to real-time units is subject to error.

• Cycle counters might be unreliable or unavailable depending on the processor or the operating system. For example, on Pentiums, the instruction is RDTSC (an assembly-language rather than a C instruction) and it is theoretically possible for the operating system to prevent user-mode programs from using it.

• Some processor details related to out-of-order execution or multiprocessor synchronization might cause the counter to seem fast or slow by up to 1000 cycles.

MySQL works with cycle counters on x386 (Windows, macOS, Linux, Solaris, and other Unix flavors), PowerPC, and IA-64.

**Performance** **Schema** **Timer** **Representation** **in** **Events**

Rows in Performance Schema tables that store current events and historical events have three columns to represent timing information: TIMER\_START and TIMER\_END indicate when an event started and finished, and TIMER\_WAIT indicates event duration.

The setup\_instruments table has an ENABLED column to indicate the instruments for which to collect events. The table also has a TIMED column to indicate which instruments are timed. If an instrument is not enabled, it produces no events. If an enabled instrument is not timed, events produced by the instrument have NULL for the TIMER\_START, TIMER\_END, and TIMER\_WAIT timer values. This in turn causes those values to be ignored when calculating aggregate time values in summary tables (sum, minimum, maximum, and average).

Internally, times within events are stored in units given by the timer in effect when event timing begins. For display when events are retrieved from Performance Schema tables, times are shown in picoseconds (trillionths of a second) to normalize them to a standard unit, regardless of which timer is selected.

The timer baseline (“time zero”) occurs at Performance Schema initialization during server startup. TIMER\_START and TIMER\_END values in events represent picoseconds since the baseline. TIMER\_WAIT values are durations in picoseconds.

Picosecond values in events are approximate. Their accuracy is subject to the usual forms of error associated with conversion from one unit to another. If the CYCLE timer is used and the processor rate varies, there might be drift. For these reasons, it is not reasonable to look at the TIMER\_START value for an event as an accurate measure of time elapsed since server startup. On the other hand, it is reasonable to use TIMER\_START or TIMER\_WAIT values in ORDER BY clauses to order events by start time or duration.

The choice of picoseconds in events rather than a value such as microseconds has a performance basis. One implementation goal was to show results in a uniform time unit, regardless of the timer. In an ideal world this time unit would look like a wall-clock unit and be reasonably precise; in other words, microseconds. But to convert cycles or nanoseconds to microseconds, it would be necessary to perform a division for every instrumentation. Division is expensive on many platforms. Multiplication is not expensive, so that is what is used. Therefore, the time unit is an integer multiple of the highest possible TIMER\_FREQUENCY value, using a multiplier large enough to ensure that there is no major precision loss. The result is that the time unit is “picoseconds.” This precision is spurious, but the decision enables overhead to be minimized.

While a wait, stage, statement, or transaction event is executing, the respective current-event tables display current-event timing information:

events\_waits\_current

events\_stages\_current

events\_statements\_current

events\_transactions\_current

To make it possible to determine how long a not-yet-completed event has been running, the timer columns are set as follows:

• TIMER\_START is populated.

• TIMER\_END is populated with the current timer value.

• TIMER\_WAIT is populated with the time elapsed so far (TIMER\_END − TIMER\_START).

Events that have not yet completed have an END\_EVENT\_ID value of NULL. To assess time elapsed so far for an event, use the TIMER\_WAIT column. Therefore, to identify events that have not yet completed and have taken longer than *N* picoseconds thus far, monitoring applications can use this expression in queries:

WHERE END\_EVENT\_ID IS NULL AND TIMER\_WAIT > *N*

Event identification as just described assumes that the corresponding instruments have ENABLED and TIMED set to YES and that the relevant consumers are enabled.

**27.4.2** **Performance** **Schema** **Event** **Filtering**

Events are processed in a producer/consumer fashion:

• Instrumented code is the source for events and produces events to be collected. The setup\_instruments table lists the instruments for which events can be collected, whether they are enabled, and (for enabled instruments) whether to collect timing information:

mysql> **SELECT** **NAME,** **ENABLED,** **TIMED**

**FROM** **performance\_schema** **.setup\_instruments;**

+---------------------------------------------------+---------+-------+

|  |  |
| --- | --- |
| | NAME | | ENABLED | TIMED | |

+---------------------------------------------------+---------+-------+

...

| wait/synch/mutex/sql/LOCK\_global\_read\_lock | YES

| wait/synch/mutex/sql/LOCK\_global\_system\_variables | YES

|

|

wait/synch/mutex/sql/LOCK\_lock\_db

wait/synch/mutex/sql/LOCK\_manager

YES

YES

YES

YES

YES

YES

|

|

|

|

|

|

|

|

|

|

...

The setup\_instruments table provides the most basic form of control over event production. To further refine event production based on the type of object or thread being monitored, other tables may be used as described in [Section 27.4.3, “Event Pre-Filtering”](#_bookmark179) .

• Performance Schema tables are the destinations for events and consume events. The

setup\_consumers table lists the types of consumers to which event information can be sent and whether they are enabled:

mysql> **SELECT** **\*** **FROM** **performance\_schema** **.setup\_consumers;**

+----------------------------------+---------+

|  |  |
| --- | --- |
| | NAME | | ENABLED | |

+----------------------------------+---------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| | | events\_stages\_current | | | NO | | |
| | | events\_stages\_history | | | NO | | |
| | | events\_stages\_history\_long | | | NO | | |
| | | events\_statements\_cpu | | | NO | | |
| | | events\_statements\_current | | | YES | | |
| | | events\_statements\_history | | | YES | | |
| | | events\_statements\_history\_long | | | NO | | |
| | | events\_transactions\_current | | | YES | | |
| | | events\_transactions\_history | | | YES | | |
| | | events\_transactions\_history\_long | | | NO | | |
| | | events\_waits\_current | | | NO | | |
| | | events\_waits\_history | | | NO | | |
| | | events\_waits\_history\_long | | | NO | | |
| | | global\_instrumentation | | | YES | | |
| | | thread\_instrumentation | | | YES | | |
| | | statements\_digest | | | YES | | |

+----------------------------------+---------+

Filtering can be done at different stages of performance monitoring:

• **Pre-filtering.** This is done by modifying Performance Schema configuration so that only certain types of events are collected from producers, and collected events update only certain consumers. To do this, enable or disable instruments or consumers. Pre-filtering is done by the Performance Schema and has a global effect that applies to all users.

Reasons to use pre-filtering:

• To reduce overhead. Performance Schema overhead should be minimal even with all instruments enabled, but perhaps you want to reduce it further. Or you do not care about timing events and want to disable the timing code to eliminate timing overhead.

• To avoid filling the current-events or history tables with events in which you have no interest. Pre- filtering leaves more “room” in these tables for instances of rows for enabled instrument types. If you enable only file instruments with pre-filtering, no rows are collected for nonfile instruments. With post-filtering, nonfile events are collected, leaving fewer rows for file events.

• To avoid maintaining some kinds of event tables. If you disable a consumer, the server does not spend time maintaining destinations for that consumer. For example, if you do not care about event histories, you can disable the history table consumers to improve performance.

• **Post-filtering.** This involves the use of WHERE clauses in queries that select information from Performance Schema tables, to specify which of the available events you want to see. Post-filtering is performed on a per-user basis because individual users select which of the available events are of interest.

Reasons to use post-filtering:

• To avoid making decisions for individual users about which event information is of interest.

• To use the Performance Schema to investigate a performance issue when the restrictions to impose using pre-filtering are not known in advance.

The following sections provide more detail about pre-filtering and provide guidelines for naming instruments or consumers in filtering operations. For information about writing queries to retrieve information (post-filtering), see Section 27.5, “Performance Schema Queries” .

**27.4.3** **Event** **Pre-Filtering**

Pre-filtering is done by the Performance Schema and has a global effect that applies to all users. Pre- filtering can be applied to either the producer or consumer stage of event processing:

• To configure pre-filtering at the producer stage, several tables can be used:

• setup\_instruments indicates which instruments are available. An instrument disabled in this table produces no events regardless of the contents of the other production-related setup tables. An instrument enabled in this table is permitted to produce events, subject to the contents of the other tables.

• setup\_objects controls whether the Performance Schema monitors particular table and stored program objects.

• threads indicates whether monitoring is enabled for each server thread.

• setup\_actors determines the initial monitoring state for new foreground threads.

• To configure pre-filtering at the consumer stage, modify the setup\_consumers table. This determines the destinations to which events are sent. setup\_consumers also implicitly affects event production. If a given event is not sent to any destination (that is, it is never consumed), the

Performance Schema does not produce it.

Modifications to any of these tables affect monitoring immediately, with the exception that modifications to the setup\_actors table affect only foreground threads created subsequent to the modification, not existing threads.

When you change the monitoring configuration, the Performance Schema does not flush the history tables. Events already collected remain in the current-events and history tables until displaced by newer events. If you disable instruments, you might need to wait a while before events for them are displaced by newer events of interest. Alternatively, use TRUNCATE TABLE to empty the history tables.

After making instrumentation changes, you might want to truncate the summary tables. Generally, the effect is to reset the summary columns to 0 or NULL, not to remove rows. This enables you to clear collected values and restart aggregation. That might be useful, for example, after you have made a runtime configuration change. Exceptions to this truncation behavior are noted in individual summary table sections.

The following sections describe how to use specific tables to control Performance Schema pre-filtering.

**27.4.4** **Pre-Filtering** **by** **Instrument**

The setup\_instruments table lists the available instruments:

mysql> **SELECT** **NAME,** **ENABLED,** **TIMED**

**FROM** **performance\_schema** **.setup\_instruments;**

+---------------------------------------------------+---------+-------+

|  |  |
| --- | --- |
| | NAME | | ENABLED | TIMED | |

+---------------------------------------------------+---------+-------+

...

|

|

|

|

...

|

|

|

|

stage/sql/end

stage/sql/executing

stage/sql/init

stage/sql/insert

statement/sql/load

statement/sql/grant

statement/sql/check

statement/sql/flush

YES

YES

YES

YES

YES

YES

YES

YES

NO

NO

NO

NO

NO

NO

NO

NO

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ...  | wait/synch/mutex/sql/LOCK\_global\_read\_lock  | wait/synch/mutex/sql/LOCK\_global\_system\_variables  | wait/synch/mutex/sql/LOCK\_lock\_db  | wait/synch/mutex/sql/LOCK\_manager  ...  | wait/synch/rwlock/sql/LOCK\_grant  | wait/synch/rwlock/sql/LOGGER::LOCK\_logger  | wait/synch/rwlock/sql/LOCK\_sys\_init\_connect  | wait/synch/rwlock/sql/LOCK\_sys\_init\_slave  ...  | wait/io/file/sql/binlog  | wait/io/file/sql/binlog\_index  | wait/io/file/sql/casetest  | wait/io/file/sql/dbopt  ... | |  |  |  |  |  |  |  |  |  |  |  | | YES  YES  YES  YES  YES  YES  YES  YES  YES  YES  YES  YES | |  |  |  |  |  |  |  |  |  |  |  | | YES  YES  YES  YES  YES  YES  YES  YES  YES  YES  YES  YES | |  |  |  |  |  |  |  |  |  |  |  | |

To control whether an instrument is enabled, set its ENABLED column to YES or NO. To configure whether to collect timing information for an enabled instrument, set its TIMED value to YES or NO. Setting the TIMED column affects Performance Schema table contents as described in [Section 27.4.1,](#_bookmark177) [“Performance Schema Event Timing”](#_bookmark177) .

Modifications to most setup\_instruments rows affect monitoring immediately. For some instruments, modifications are effective only at server startup; changing them at runtime has no effect. This affects primarily mutexes, conditions, and rwlocks in the server, although there may be other instruments for which this is true.

The setup\_instruments table provides the most basic form of control over event production. To further refine event production based on the type of object or thread being monitored, other tables may be used as described in [Section 27.4.3, “Event Pre-Filtering”](#_bookmark179) .

The following examples demonstrate possible operations on the setup\_instruments table. These changes, like other pre-filtering operations, affect all users. Some of these queries use the LIKE operator and a pattern match instrument names. For additional information about specifying patterns to select instruments, see Section 27.4.9, “Naming Instruments or Consumers for Filtering Operations” .

• Disable all instruments:

UPDATE performance\_schema.setup\_instruments

SET ENABLED = 'NO';

Now no events are collected.

• Disable all file instruments, adding them to the current set of disabled instruments:

UPDATE performance\_schema.setup\_instruments

SET ENABLED = 'NO'

WHERE NAME LIKE 'wait/io/file/%';

• Disable only file instruments, enable all other instruments:

UPDATE performance\_schema.setup\_instruments

SET ENABLED = IF(NAME LIKE 'wait/io/file/%', 'NO', 'YES');

• Enable all but those instruments in the mysys library:

UPDATE performance\_schema.setup\_instruments

SET ENABLED = CASE WHEN NAME LIKE '%/mysys/%' THEN 'YES' ELSE 'NO' END;

• Disable a specific instrument:

UPDATE performance\_schema.setup\_instruments

SET ENABLED = 'NO'

WHERE NAME = 'wait/synch/mutex/mysys/TMPDIR\_mutex';

• To toggle the state of an instrument, “flip” its ENABLED value:

UPDATE performance\_schema.setup\_instruments

SET ENABLED = IF(ENABLED = 'YES', 'NO', 'YES')

WHERE NAME = 'wait/synch/mutex/mysys/TMPDIR\_mutex';

• Disable timing for all events:

UPDATE performance\_schema.setup\_instruments

SET TIMED = 'NO';

**27.4.5** **Pre-Filtering** **by** **Object**

The setup\_objects table controls whether the Performance Schema monitors particular table and stored program objects. The initial setup\_objects contents look like this:

mysql> **SELECT** **\*** **FROM** **performance\_schema** **.setup\_objects;**

+-------------+--------------------+-------------+---------+-------+

| OBJECT\_TYPE | OBJECT\_SCHEMA | OBJECT\_NAME | ENABLED | TIMED |

+-------------+--------------------+-------------+---------+-------+

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  | | EVENT  EVENT  EVENT  EVENT  FUNCTION  FUNCTION  FUNCTION  FUNCTION  PROCEDURE  PROCEDURE  PROCEDURE  PROCEDURE  TABLE  TABLE  TABLE  TABLE  TRIGGER  TRIGGER  TRIGGER  TRIGGER | |  |  |  |  |  |  |  |  |  |  |  | | mysql  performance\_schema  information\_schema  %  mysql  performance\_schema  information\_schema  %  mysql  performance\_schema  information\_schema  %  mysql  performance\_schema  information\_schema  %  mysql  performance\_schema  information\_schema  % |

+-------------+--------------------+-------------+---------+-------+

NO

NO

NO

YES

NO

NO

NO

YES

NO

NO

NO

YES

NO

NO

NO

YES

NO

NO

NO

YES

NO

NO

NO

YES

NO

NO

NO

YES

NO

NO

NO

YES

NO

NO

NO

YES

NO

NO

NO

YES

%

%

%

%

%

%

%

%

%

%

%

%

%

%

%

%

%

%

%

%

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

Modifications to the setup\_objects table affect object monitoring immediately.

The OBJECT\_TYPE column indicates the type of object to which a row applies. TABLE filtering affects table I/O events (wait/io/table/sql/handler instrument) and table lock events (wait/lock/ table/sql/handler instrument).

The OBJECT\_SCHEMA and OBJECT\_NAME columns should contain a literal schema or object name, or '%' to match any name.

The ENABLED column indicates whether matching objects are monitored, and TIMED indicates whether to collect timing information. Setting the TIMED column affects Performance Schema table contents as described in [Section 27.4.1, “Performance Schema Event Timing”](#_bookmark177) .

The effect of the default object configuration is to instrument all objects except those in the mysql, INFORMATION\_SCHEMA, and performance\_schema databases. (Tables in the INFORMATION\_SCHEMA database are not instrumented regardless of the contents of setup\_objects; the row for information\_schema.% simply makes this default explicit.)

When the Performance Schema checks for a match in setup\_objects, it tries to find more specific matches first. For rows that match a given OBJECT\_TYPE, the Performance Schema checks rows in this order:

• Rows with OBJECT\_SCHEMA='*literal*' and OBJECT\_NAME='*literal*'.

• Rows with OBJECT\_SCHEMA='*literal*' and OBJECT\_NAME='%'.

• Rows with OBJECT\_SCHEMA='%' and OBJECT\_NAME='%'.

For example, with a table db1.t1, the Performance Schema looks in TABLE rows for a match for 'db1' and 't1', then for 'db1' and '%', then for '%' and '%'. The order in which matching occurs matters because different matching setup\_objects rows can have different ENABLED and TIMED values.

For table-related events, the Performance Schema combines the contents of setup\_objects with setup\_instruments to determine whether to enable instruments and whether to time enabled instruments:

• For tables that match a row in setup\_objects, table instruments produce events only if ENABLED is YES in both setup\_instruments and setup\_objects.

• The TIMED values in the two tables are combined, so that timing information is collected only when both values are YES.

For stored program objects, the Performance Schema takes the ENABLED and TIMED columns directly from the setup\_objects row. There is no combining of values with setup\_instruments.

Suppose that setup\_objects contains the following TABLE rows that apply to db1, db2, and db3:

+-------------+---------------+-------------+---------+-------+

| OBJECT\_TYPE | OBJECT\_SCHEMA | OBJECT\_NAME | ENABLED | TIMED |

+-------------+---------------+-------------+---------+-------+

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | TABLE | | | db1 | | | t1 | | | YES | | | YES | | |
| | | TABLE | | | db1 | | | t2 | | | NO | | | NO | | |
| | | TABLE | | | db2 | | | % | | | YES | | | YES | | |
| | | TABLE | | | db3 | | | % | | | NO | | | NO | | |
| | | TABLE | | | % | | | % | | | YES | | | YES | | |

+-------------+---------------+-------------+---------+-------+

If an object-related instrument in setup\_instruments has an ENABLED value of NO, events for the object are not monitored. If the ENABLED value is YES, event monitoring occurs according to the ENABLED value in the relevant setup\_objects row:

• db1.t1 events are monitored

• db1.t2 events are not monitored

• db2.t3 events are monitored

• db3.t4 events are not monitored

• db4.t5 events are monitored

Similar logic applies for combining the TIMED columns from the setup\_instruments and setup\_objects tables to determine whether to collect event timing information.

If a persistent table and a temporary table have the same name, matching against setup\_objects rows occurs the same way for both. It is not possible to enable monitoring for one table but not the other. However, each table is instrumented separately.

**27.4.6** **Pre-Filtering** **by** **Thread**

The threads table contains a row for each server thread. Each row contains information about a thread and indicates whether monitoring is enabled for it. For the Performance Schema to monitor a thread, these things must be true:

• The thread\_instrumentation consumer in the setup\_consumers table must be YES.

• The threads.INSTRUMENTED column must be YES.

• Monitoring occurs only for those thread events produced from instruments that are enabled in the setup\_instruments table.

The threads table also indicates for each server thread whether to perform historical event logging. This includes wait, stage, statement, and transaction events and affects logging to these tables:

events\_waits\_history

events\_waits\_history\_long

events\_stages\_history

events\_stages\_history\_long

events\_statements\_history

events\_statements\_history\_long

events\_transactions\_history

events\_transactions\_history\_long

For historical event logging to occur, these things must be true:

• The appropriate history-related consumers in the setup\_consumers table must be enabled. For example, wait event logging in the events\_waits\_history and events\_waits\_history\_long tables requires the corresponding events\_waits\_history and events\_waits\_history\_long consumers to be YES.

• The threads.HISTORY column must be YES.

• Logging occurs only for those thread events produced from instruments that are enabled in the setup\_instruments table.

For foreground threads (resulting from client connections), the initial values of the INSTRUMENTED and HISTORY columns in threads table rows are determined by whether the user account associated with a thread matches any row in the setup\_actors table. The values come from the ENABLED and HISTORY columns of the matching setup\_actors table row.

For background threads, there is no associated user. INSTRUMENTED and HISTORY are YES by default

and setup\_actors is not consulted.

The initial setup\_actors contents look like this:

mysql> **SELECT** **\*** **FROM** **performance\_schema** **.setup\_actors;**

+------+------+------+---------+---------+

| HOST | USER | ROLE | ENABLED | HISTORY |

+------+------+------+---------+---------+

| % | % | % | YES | YES |

+------+------+------+---------+---------+

The HOST and USER columns should contain a literal host or user name, or '%' to match any name.

The ENABLED and HISTORY columns indicate whether to enable instrumentation and historical event logging for matching threads, subject to the other conditions described previously.

When the Performance Schema checks for a match for each new foreground thread in setup\_actors, it tries to find more specific matches first, using the USER and HOST columns (ROLE is unused):

• Rows with USER='*literal*' and HOST='*literal*'.

• Rows with USER='*literal*' and HOST='%'.

• Rows with USER='%' and HOST='*literal*'.

• Rows with USER='%' and HOST='%'.

The order in which matching occurs matters because different matching setup\_actors rows can have different USER and HOST values. This enables instrumenting and historical event logging to be applied selectively per host, user, or account (user and host combination), based on the ENABLED and HISTORY column values:

• When the best match is a row with ENABLED=YES, the INSTRUMENTED value for the thread becomes YES. When the best match is a row with HISTORY=YES, the HISTORY value for the thread becomes YES.

• When the best match is a row with ENABLED=NO, the INSTRUMENTED value for the thread becomes NO. When the best match is a row with HISTORY=NO, the HISTORY value for the thread becomes NO.