MySQL Performance Schema

Abstract

This is the MySQL Performance Schema extract from the MySQL 8.0 Reference Manual.

For help with using MySQL, please visit the MySQL Forums, where you can discuss your issues with other MySQL users.

Document generated on: 2025-04-28 (revision: 81812)

Table of Contents

Preface and Legal Notices

This is the MySQL Performance Schema extract from the MySQL 8.0 Reference Manual.

Licensing information—MySQL 8.0. This product may include third-party software, used under license. If you are using a *Commercial* release of MySQL 8.0, see the MySQL 8.0 Commercial Release License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using a *Community* release of MySQL 8.0, see the MySQL 8.0 Community Release License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Legal Notices

License Restrictions

Warranty Disclaimer

Restricted Rights Notice

Hazardous Applications Notice

measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Trademark Notice

Oracle, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

Third-Party Content, Products, and Services Disclaimer

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Use of This Documentation

This documentation is NOT distributed under a GPL license. Use of this documentation is subject to the following terms:

You may create a printed copy of this documentation solely for your own personal use. Conversion to other formats is allowed as long as the actual content is not altered or edited in any way. You shall not publish or distribute this documentation in any form or on any media, except if you distribute the documentation in a manner similar to how Oracle disseminates it (that is, electronically for download on a Web site with the software) or on a CD-ROM or similar medium, provided however that the documentation is disseminated together with the software on the same medium. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Oracle. Oracle and/or its affiliates reserve any and all rights to this documentation not expressly granted above.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Chapter 1 MySQL Performance Schema

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` 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` 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 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 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 MySQL sys Schema.

Chapter 2 Performance Schema Quick Start

This section briefly introduces the Performance Schema with examples that show how to use it. For additional examples, see Chapter 14, *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` 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` 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` 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 Chapter 10, *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 5.1, "Performance Schema Event Timing".

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        |       6419 |
| wait/io/file/sql/FRM                          |        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                                      | NO      | NO    |
| stage/sql/executing                                | NO      | NO    |
| stage/sql/init                                     | NO      | NO    |
| stage/sql/insert                                   | NO      | NO    |
...
| statement/sql/load                                 | YES     | YES   |
| statement/sql/grant                                | YES     | YES   |
| statement/sql/check                                | YES     | YES   |
| statement/sql/flush                                | YES     | YES   |
...
```

```
| wait/synch/mutex/sql/LOCK_global_read_lock         | YES     | YES   |
| wait/synch/mutex/sql/LOCK_global_system_variables | YES     | YES   |
| wait/synch/mutex/sql/LOCK_lock_db                 | YES     | YES   |
| wait/synch/mutex/sql/LOCK_manager                 | YES     | YES   |
...
| wait/synch/rwlock/sql/LOCK_grant                  | YES     | YES   |
| wait/synch/rwlock/sql/LOGGER::LOCK_logger         | YES     | YES   |
| wait/synch/rwlock/sql/LOCK_sys_init_connect       | YES     | YES   |
| wait/synch/rwlock/sql/LOCK_sys_init_slave         | YES     | YES   |
...
| wait/io/file/sql/binlog                           | YES     | YES   |
| wait/io/file/sql/binlog_index                     | YES     | YES   |
| wait/io/file/sql/casetest                         | YES     | YES   |
| wait/io/file/sql/dbopt                            | YES     | YES   |
...
```

To understand how to interpret instrument names, see Chapter 7, *Performance Schema Instrument Naming Conventions*.

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;
+----------------------------------+---------+
| 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     |
+----------------------------------+---------+
```

To control whether the Performance Schema maintains a consumer as a destination for event information, set its `ENABLED` value.

For more information about the setup tables and how to use them to control event collection, see Section 5.2, “Performance Schema Event Filtering”.

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 5.1, “Performance Schema Event Timing”.

Chapter 3 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 \
        -DDISABLE_PSI_STATEMENT=1
```

For more information, see the descriptions of the `DISABLE_PSI_XXX` `CMake` options in 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 Upgrading MySQL.

Because the Performance Schema is configured into the server at build time, a row for `PERFORMANCE_SCHEMA` 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.

Chapter 4 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 5.9, “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 required; see 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 Chapter 12, *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.

Chapter 5 Performance Schema Runtime Configuration

Table of Contents

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'
       AND TABLE_NAME LIKE 'setup%';
+-------------------+
| TABLE_NAME        |
+-------------------+
| setup_actors      |
| setup_consumers   |
| setup_instruments |
| setup_objects     |
| setup_threads     |
+-------------------+
```

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 10.2, “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 5.2, “Performance Schema Event Filtering”, 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.

5.1 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.

5.2 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     | YES   |
| wait/synch/mutex/sql/LOCK_global_system_variables | YES     | YES   |
| wait/synch/mutex/sql/LOCK_lock_db                  | YES     | YES   |
| wait/synch/mutex/sql/LOCK_manager                  | 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 5.3, “Event Pre-Filtering”.

- 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 Chapter 6, *Performance Schema Queries*.

5.3 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.

5.4 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                                     | NO      | NO    |
| stage/sql/executing                               | NO      | NO    |
| stage/sql/init                                    | NO      | NO    |
| stage/sql/insert                                  | NO      | NO    |
...
| statement/sql/load                                | YES     | YES   |
| statement/sql/grant                               | YES     | YES   |
```

```
| statement/sql/check                               | YES     | YES   |
| statement/sql/flush                               | YES     | YES   |
...
| wait/synch/mutex/sql/LOCK_global_read_lock        | YES     | YES   |
| wait/synch/mutex/sql/LOCK_global_system_variables | YES     | YES   |
| wait/synch/mutex/sql/LOCK_lock_db                 | YES     | YES   |
| wait/synch/mutex/sql/LOCK_manager                 | YES     | YES   |
...
| wait/synch/rwlock/sql/LOCK_grant                  | YES     | YES   |
| wait/synch/rwlock/sql/LOGGER::LOCK_logger         | YES     | YES   |
| wait/synch/rwlock/sql/LOCK_sys_init_connect       | YES     | YES   |
| wait/synch/rwlock/sql/LOCK_sys_init_slave         | YES     | YES   |
...
| wait/io/file/sql/binlog                           | YES     | YES   |
| wait/io/file/sql/binlog_index                     | YES     | YES   |
| wait/io/file/sql/casetest                         | YES     | YES   |
| wait/io/file/sql/dbopt                            | 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 5.1, “Performance Schema Event Timing”.

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 5.3, “Event Pre-Filtering”.

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 5.9, “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';
```

5.5 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       | mysql              | %           | NO      | NO    |
| EVENT       | performance_schema | %           | NO      | NO    |
| EVENT       | information_schema | %           | NO      | NO    |
| EVENT       | %                  | %           | YES     | YES   |
| FUNCTION    | mysql              | %           | NO      | NO    |
| FUNCTION    | performance_schema | %           | NO      | NO    |
| FUNCTION    | information_schema | %           | NO      | NO    |
| FUNCTION    | %                  | %           | YES     | YES   |
| PROCEDURE   | mysql              | %           | NO      | NO    |
| PROCEDURE   | performance_schema | %           | NO      | NO    |
| PROCEDURE   | information_schema | %           | NO      | NO    |
| PROCEDURE   | %                  | %           | YES     | YES   |
| TABLE       | mysql              | %           | NO      | NO    |
| TABLE       | performance_schema | %           | NO      | NO    |
| TABLE       | information_schema | %           | NO      | NO    |
| TABLE       | %                  | %           | YES     | YES   |
| TRIGGER     | mysql              | %           | NO      | NO    |
| TRIGGER     | performance_schema | %           | NO      | NO    |
| TRIGGER     | information_schema | %           | NO      | NO    |
| TRIGGER     | %                  | %           | YES     | 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 5.1, “Performance Schema Event Timing”.

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.

5.6 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.
- When no match is found, the INSTRUMENTED and HISTORY values for the thread become NO.

The ENABLED and HISTORY columns in setup_actors rows can be set to YES or NO independent of one another. This means you can enable instrumentation separately from whether you collect historical events.

By default, monitoring and historical event collection are enabled for all new foreground threads because the setup_actors table initially contains a row with '%' for both HOST and USER. To perform more limited matching such as to enable monitoring only for some foreground threads, you must change this row because it matches any connection, and add rows for more specific HOST/USER combinations.

Suppose that you modify setup_actors as follows:

```
UPDATE performance_schema.setup_actors
SET ENABLED = 'NO', HISTORY = 'NO'
WHERE HOST = '%' AND USER = '%';
INSERT INTO performance_schema.setup_actors
(HOST,USER,ROLE,ENABLED,HISTORY)
VALUES('localhost','joe','%','YES','YES');
INSERT INTO performance_schema.setup_actors
(HOST,USER,ROLE,ENABLED,HISTORY)
VALUES('hosta.example.com','joe','%','YES','NO');
INSERT INTO performance_schema.setup_actors
(HOST,USER,ROLE,ENABLED,HISTORY)
VALUES('%','sam','%','NO','YES');
```

The UPDATE statement changes the default match to disable instrumentation and historical event collection. The INSERT statements add rows for more specific matches.

Now the Performance Schema determines how to set the INSTRUMENTED and HISTORY values for new connection threads as follows:

- If joe connects from the local host, the connection matches the first inserted row. The INSTRUMENTED and HISTORY values for the thread become YES.

- If `joe` connects from `hosta.example.com`, the connection matches the second inserted row. The `INSTRUMENTED` value for the thread becomes `YES` and the `HISTORY` value becomes `NO`.
- If `joe` connects from any other host, there is no match. The `INSTRUMENTED` and `HISTORY` values for the thread become `NO`.
- If `sam` connects from any host, the connection matches the third inserted row. The `INSTRUMENTED` value for the thread becomes `NO` and the `HISTORY` value becomes `YES`.
- For any other connection, the row with `HOST` and `USER` set to `'%'` matches. This row now has `ENABLED` and `HISTORY` set to `NO`, so the `INSTRUMENTED` and `HISTORY` values for the thread become `NO`.

Modifications to the `setup_actors` table affect only foreground threads created subsequent to the modification, not existing threads. To affect existing threads, modify the `INSTRUMENTED` and `HISTORY` columns of `threads` table rows.

5.7 Pre-Filtering by Consumer

The `setup_consumers` table lists the available consumer types and which 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     |
+--------------------------------+---------+
```

Modify the `setup_consumers` table to affect pre-filtering at the consumer stage and determine the destinations to which events are sent. To enable or disable a consumer, set its `ENABLED` value to `YES` or `NO`.

Modifications to the `setup_consumers` table affect monitoring immediately.

If you disable a consumer, the server does not spend time maintaining destinations for that consumer. For example, if you do not care about historical event information, disable the history consumers:

```
UPDATE performance_schema.setup_consumers
SET ENABLED = 'NO'
WHERE NAME LIKE '%history%';
```

The consumer settings in the `setup_consumers` table form a hierarchy from higher levels to lower. The following principles apply:

- Destinations associated with a consumer receive no events unless the Performance Schema checks the consumer and the consumer is enabled.

- A consumer is checked only if all consumers it depends on (if any) are enabled.
- If a consumer is not checked, or is checked but is disabled, other consumers that depend on it are not checked.
- Dependent consumers may have their own dependent consumers.
- If an event would not be sent to any destination, the Performance Schema does not produce it.

The following lists describe the available consumer values. For discussion of several representative consumer configurations and their effect on instrumentation, see Section 5.8, “Example Consumer Configurations”.

- Global and Thread Consumers
- Wait Event Consumers
- Stage Event Consumers
- Statement Event Consumers
- Transaction Event Consumers
- Statement Digest Consumer

Global and Thread Consumers

- `global_instrumentation` is the highest level consumer. If `global_instrumentation` is `NO`, it disables global instrumentation. All other settings are lower level and are not checked; it does not matter what they are set to. No global or per thread information is maintained and no individual events are collected in the current-events or event-history tables. If `global_instrumentation` is `YES`, the Performance Schema maintains information for global states and also checks the `thread_instrumentation` consumer.
- `thread_instrumentation` is checked only if `global_instrumentation` is `YES`. Otherwise, if `thread_instrumentation` is `NO`, it disables thread-specific instrumentation and all lower-level settings are ignored. No information is maintained per thread and no individual events are collected in the current-events or event-history tables. If `thread_instrumentation` is `YES`, the Performance Schema maintains thread-specific information and also checks `events_xxx_current` consumers.

Wait Event Consumers

These consumers require both `global_instrumentation` and `thread_instrumentation` to be `YES` or they are not checked. If checked, they act as follows:

- `events_waits_current`, if `NO`, disables collection of individual wait events in the `events_waits_current` table. If `YES`, it enables wait event collection and the Performance Schema checks the `events_waits_history` and `events_waits_history_long` consumers.
- `events_waits_history` is not checked if `event_waits_current` is `NO`. Otherwise, an `events_waits_history` value of `NO` or `YES` disables or enables collection of wait events in the `events_waits_history` table.
- `events_waits_history_long` is not checked if `event_waits_current` is `NO`. Otherwise, an `events_waits_history_long` value of `NO` or `YES` disables or enables collection of wait events in the `events_waits_history_long` table.

Stage Event Consumers

These consumers require both `global_instrumentation` and `thread_instrumentation` to be `YES` or they are not checked. If checked, they act as follows:

- `events_stages_current`, if `NO`, disables collection of individual stage events in the `events_stages_current` table. If `YES`, it enables stage event collection and the Performance Schema checks the `events_stages_history` and `events_stages_history_long` consumers.
- `events_stages_history` is not checked if `event_stages_current` is `NO`. Otherwise, an `events_stages_history` value of `NO` or `YES` disables or enables collection of stage events in the `events_stages_history` table.
- `events_stages_history_long` is not checked if `event_stages_current` is `NO`. Otherwise, an `events_stages_history_long` value of `NO` or `YES` disables or enables collection of stage events in the `events_stages_history_long` table.

Statement Event Consumers

These consumers require both `global_instrumentation` and `thread_instrumentation` to be `YES` or they are not checked. If checked, they act as follows:

- `events_statements_cpu`, if `NO`, disables measurement of `CPU_TIME`. If `YES`, and the instrumentation is enabled and timed, `CPU_TIME` is measured.
- `events_statements_current`, if `NO`, disables collection of individual statement events in the `events_statements_current` table. If `YES`, it enables statement event collection and the Performance Schema checks the `events_statements_history` and `events_statements_history_long` consumers.
- `events_statements_history` is not checked if `events_statements_current` is `NO`. Otherwise, an `events_statements_history` value of `NO` or `YES` disables or enables collection of statement events in the `events_statements_history` table.
- `events_statements_history_long` is not checked if `events_statements_current` is `NO`. Otherwise, an `events_statements_history_long` value of `NO` or `YES` disables or enables collection of statement events in the `events_statements_history_long` table.

Transaction Event Consumers

These consumers require both `global_instrumentation` and `thread_instrumentation` to be `YES` or they are not checked. If checked, they act as follows:

- `events_transactions_current`, if `NO`, disables collection of individual transaction events in the `events_transactions_current` table. If `YES`, it enables transaction event collection and the Performance Schema checks the `events_transactions_history` and `events_transactions_history_long` consumers.
- `events_transactions_history` is not checked if `events_transactions_current` is `NO`. Otherwise, an `events_transactions_history` value of `NO` or `YES` disables or enables collection of transaction events in the `events_transactions_history` table.
- `events_transactions_history_long` is not checked if `events_transactions_current` is `NO`. Otherwise, an `events_transactions_history_long` value of `NO` or `YES` disables or enables collection of transaction events in the `events_transactions_history_long` table.

Statement Digest Consumer

The `statements_digest` consumer requires `global_instrumentation` to be `YES` or it is not checked. There is no dependency on the statement event consumers, so you can obtain statistics per digest without having to collect statistics in `events_statements_current`, which is advantageous in terms of overhead. Conversely, you can get detailed statements in `events_statements_current` without digests (the `DIGEST` and `DIGEST_TEXT` columns are `NULL` in this case).

For more information about statement digesting, see Performance Schema Statement Digests and Sampling.

5.8 Example Consumer Configurations

The consumer settings in the `setup_consumers` table form a hierarchy from higher levels to lower. The following discussion describes how consumers work, showing specific configurations and their effects as consumer settings are enabled progressively from high to low. The consumer values shown are representative. The general principles described here apply to other consumer values that may be available.

The configuration descriptions occur in order of increasing functionality and overhead. If you do not need the information provided by enabling lower-level settings, disable them so that the Performance Schema executes less code on your behalf and there is less information to sift through.

The `setup_consumers` table contains the following hierarchy of values:

```
global_instrumentation
 thread_instrumentation
   events_waits_current
     events_waits_history
     events_waits_history_long
   events_stages_current
     events_stages_history
     events_stages_history_long
   events_statements_current
     events_statements_history
     events_statements_history_long
   events_transactions_current
     events_transactions_history
     events_transactions_history_long
 statements_digest
```

> **Note**
>
> In the consumer hierarchy, the consumers for waits, stages, statements, and transactions are all at the same level. This differs from the event nesting hierarchy, for which wait events nest within stage events, which nest within statement events, which nest within transaction events.

If a given consumer setting is `NO`, the Performance Schema disables the instrumentation associated with the consumer and ignores all lower-level settings. If a given setting is `YES`, the Performance Schema enables the instrumentation associated with it and checks the settings at the next lowest level. For a description of the rules for each consumer, see Section 5.7, “Pre-Filtering by Consumer”.

For example, if `global_instrumentation` is enabled, `thread_instrumentation` is checked. If `thread_instrumentation` is enabled, the `events_xxx_current` consumers are checked. If of these `events_waits_current` is enabled, `events_waits_history` and `events_waits_history_long` are checked.

Each of the following configuration descriptions indicates which setup elements the Performance Schema checks and which output tables it maintains (that is, for which tables it collects information).

- No Instrumentation
- Global Instrumentation Only
- Global and Thread Instrumentation Only
- Global, Thread, and Current-Event Instrumentation
- Global, Thread, Current-Event, and Event-History instrumentation

No Instrumentation

Server configuration state:

```
mysql> SELECT * FROM performance_schema.setup_consumers;
+---------------------------+---------+
| NAME                      | ENABLED |
+---------------------------+---------+
| global_instrumentation    | NO      |
...
+---------------------------+---------+
```

In this configuration, nothing is instrumented.

Setup elements checked:

- Table `setup_consumers`, consumer `global_instrumentation`

Output tables maintained:

- None

Global Instrumentation Only

Server configuration state:

```
mysql> SELECT * FROM performance_schema.setup_consumers;
+---------------------------+---------+
| NAME                      | ENABLED |
+---------------------------+---------+
| global_instrumentation    | YES     |
| thread_instrumentation    | NO      |
...
+---------------------------+---------+
```

In this configuration, instrumentation is maintained only for global states. Per-thread instrumentation is disabled.

Additional setup elements checked, relative to the preceding configuration:

- Table `setup_consumers`, consumer `thread_instrumentation`
- Table `setup_instruments`
- Table `setup_objects`

Additional output tables maintained, relative to the preceding configuration:

- mutex_instances
- rwlock_instances
- cond_instances
- file_instances
- users
- hosts
- accounts
- socket_summary_by_event_name
- file_summary_by_instance
- file_summary_by_event_name
- objects_summary_global_by_type
- memory_summary_global_by_event_name
- table_lock_waits_summary_by_table
- table_io_waits_summary_by_index_usage
- table_io_waits_summary_by_table
- events_waits_summary_by_instance
- events_waits_summary_global_by_event_name
- events_stages_summary_global_by_event_name
- events_statements_summary_global_by_event_name
- events_transactions_summary_global_by_event_name

Global and Thread Instrumentation Only

Server configuration state:

```
mysql> SELECT * FROM performance_schema.setup_consumers;
+----------------------------------+---------+
| NAME                             | ENABLED |
+----------------------------------+---------+
| global_instrumentation           | YES     |
| thread_instrumentation           | YES     |
| events_waits_current             | NO      |
...
| events_stages_current            | NO      |
...
| events_statements_current        | NO      |
...
| events_transactions_current      | NO      |
...
+----------------------------------+---------+
```

In this configuration, instrumentation is maintained globally and per thread. No individual events are collected in the current-events or event-history tables.

Additional setup elements checked, relative to the preceding configuration:

- Table `setup_consumers`, consumers `events_xxx_current`, where *`xxx`* is `waits`, `stages`, `statements`, `transactions`
- Table `setup_actors`
- Column `threads.instrumented`

Additional output tables maintained, relative to the preceding configuration:

- `events_xxx_summary_by_yyy_by_event_name`, where *`xxx`* is `waits`, `stages`, `statements`, `transactions`; and *`yyy`* is `thread`, `user`, `host`, `account`

Global, Thread, and Current-Event Instrumentation

Server configuration state:

```
mysql> SELECT * FROM performance_schema.setup_consumers;
+--------------------------------+---------+
| NAME                           | ENABLED |
+--------------------------------+---------+
| global_instrumentation         | YES     |
| thread_instrumentation         | YES     |
| events_waits_current           | YES     |
| events_waits_history           | NO      |
| events_waits_history_long      | NO      |
| events_stages_current          | YES     |
| events_stages_history          | NO      |
| events_stages_history_long     | NO      |
| events_statements_current      | YES     |
| events_statements_history      | NO      |
| events_statements_history_long | NO      |
| events_transactions_current    | YES     |
| events_transactions_history    | NO      |
| events_transactions_history_long | NO    |
...
+--------------------------------+---------+
```

In this configuration, instrumentation is maintained globally and per thread. Individual events are collected in the current-events table, but not in the event-history tables.

Additional setup elements checked, relative to the preceding configuration:

- Consumers `events_xxx_history`, where *`xxx`* is `waits`, `stages`, `statements`, `transactions`
- Consumers `events_xxx_history_long`, where *`xxx`* is `waits`, `stages`, `statements`, `transactions`

Additional output tables maintained, relative to the preceding configuration:

- `events_xxx_current`, where *`xxx`* is `waits`, `stages`, `statements`, `transactions`

Global, Thread, Current-Event, and Event-History instrumentation

The preceding configuration collects no event history because the `events_xxx_history` and `events_xxx_history_long` consumers are disabled. Those consumers can be enabled separately or together to collect event history per thread, globally, or both.

This configuration collects event history per thread, but not globally:

```
mysql> SELECT * FROM performance_schema.setup_consumers;
+--------------------------------+---------+
| NAME                           | ENABLED |
+--------------------------------+---------+
| global_instrumentation         | YES     |
| thread_instrumentation         | YES     |
| events_waits_current           | YES     |
| events_waits_history           | YES     |
| events_waits_history_long      | NO      |
| events_stages_current          | YES     |
| events_stages_history          | YES     |
| events_stages_history_long     | 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    |
...
+--------------------------------+---------+
```

Event-history tables maintained for this configuration:

- `events_xxx_history`, where *xxx* is `waits`, `stages`, `statements`, `transactions`

This configuration collects event history globally, but not per thread:

```
mysql> SELECT * FROM performance_schema.setup_consumers;
+--------------------------------+---------+
| NAME                           | ENABLED |
+--------------------------------+---------+
| global_instrumentation         | YES     |
| thread_instrumentation         | YES     |
| events_waits_current           | YES     |
| events_waits_history           | NO      |
| events_waits_history_long      | YES     |
| events_stages_current          | YES     |
| events_stages_history          | NO      |
| events_stages_history_long     | YES     |
| events_statements_current      | YES     |
| events_statements_history      | NO      |
| events_statements_history_long | YES     |
| events_transactions_current    | YES     |
| events_transactions_history    | NO      |
| events_transactions_history_long | YES   |
...
+--------------------------------+---------+
```

Event-history tables maintained for this configuration:

- `events_xxx_history_long`, where *xxx* is `waits`, `stages`, `statements`, `transactions`

This configuration collects event history per thread and globally:

```
mysql> SELECT * FROM performance_schema.setup_consumers;
+--------------------------------+---------+
| NAME                           | ENABLED |
+--------------------------------+---------+
| global_instrumentation         | YES     |
| thread_instrumentation         | YES     |
| events_waits_current           | YES     |
| events_waits_history           | YES     |
| events_waits_history_long      | YES     |
| events_stages_current          | YES     |
| events_stages_history          | YES     |
| events_stages_history_long     | YES     |
```

```
| events_statements_current        | YES     |
| events_statements_history        | YES     |
| events_statements_history_long   | YES     |
| events_transactions_current      | YES     |
| events_transactions_history      | YES     |
| events_transactions_history_long | YES     |
...
+----------------------------------+---------+
```

Event-history tables maintained for this configuration:

- `events_xxx_history`, where *xxx* is `waits`, `stages`, `statements`, `transactions`
- `events_xxx_history_long`, where *xxx* is `waits`, `stages`, `statements`, `transactions`

5.9 Naming Instruments or Consumers for Filtering Operations

Names given for filtering operations can be as specific or general as required. To indicate a single instrument or consumer, specify its name in full:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'NO'
WHERE NAME = 'wait/synch/mutex/myisammrg/MYRG_INFO::mutex';
UPDATE performance_schema.setup_consumers
SET ENABLED = 'NO'
WHERE NAME = 'events_waits_current';
```

To specify a group of instruments or consumers, use a pattern that matches the group members:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'NO'
WHERE NAME LIKE 'wait/synch/mutex/%';
UPDATE performance_schema.setup_consumers
SET ENABLED = 'NO'
WHERE NAME LIKE '%history%';
```

If you use a pattern, it should be chosen so that it matches all the items of interest and no others. For example, to select all file I/O instruments, it is better to use a pattern that includes the entire instrument name prefix:

```
... WHERE NAME LIKE 'wait/io/file/%';
```

A pattern of `'%/file/%'` matches other instruments that have an element of `'/file/'` anywhere in the name. Even less suitable is the pattern `'%file%'` because it matches instruments with `'file'` anywhere in the name, such as `wait/synch/mutex/innodb/file_open_mutex`.

To check which instrument or consumer names a pattern matches, perform a simple test:

```
SELECT NAME FROM performance_schema.setup_instruments
WHERE NAME LIKE 'pattern';
SELECT NAME FROM performance_schema.setup_consumers
WHERE NAME LIKE 'pattern';
```

For information about the types of names that are supported, see Chapter 7, *Performance Schema Instrument Naming Conventions*.

5.10 Determining What Is Instrumented

It is always possible to determine what instruments the Performance Schema includes by checking the `setup_instruments` table. For example, to see what file-related events are instrumented for the `InnoDB` storage engine, use this query:

```
mysql> SELECT NAME, ENABLED, TIMED
       FROM performance_schema.setup_instruments
       WHERE NAME LIKE 'wait/io/file/innodb/%';
+----------------------------------------------------+---------+-------+
| NAME                                               | ENABLED | TIMED |
+----------------------------------------------------+---------+-------+
| wait/io/file/innodb/innodb_tablespace_open_file | YES     | YES   |
| wait/io/file/innodb/innodb_data_file            | YES     | YES   |
| wait/io/file/innodb/innodb_log_file             | YES     | YES   |
| wait/io/file/innodb/innodb_temp_file            | YES     | YES   |
| wait/io/file/innodb/innodb_arch_file            | YES     | YES   |
| wait/io/file/innodb/innodb_clone_file           | YES     | YES   |
+----------------------------------------------------+---------+-------+
```

An exhaustive description of precisely what is instrumented is not given in this documentation, for several reasons:

- What is instrumented is the server code. Changes to this code occur often, which also affects the set of instruments.
- It is not practical to list all the instruments because there are hundreds of them.
- As described earlier, it is possible to find out by querying the `setup_instruments` table. This information is always up to date for your version of MySQL, also includes instrumentation for instrumented plugins you might have installed that are not part of the core server, and can be used by automated tools.

Pre-filtering limits which event information is collected and is independent of any particular user. By contrast, post-filtering is performed by individual users through the use of queries with appropriate `WHERE` clauses that restrict what event information to select from the events available after pre-filtering has been applied.

In Section 5.3, “Event Pre-Filtering”, an example showed how to pre-filter for file instruments. If the event tables contain both file and nonfile information, post-filtering is another way to see information only for file events. Add a `WHERE` clause to queries to restrict event selection appropriately:

```
mysql> SELECT THREAD_ID, NUMBER_OF_BYTES
       FROM performance_schema.events_waits_history
       WHERE EVENT_NAME LIKE 'wait/io/file/%'
       AND NUMBER_OF_BYTES IS NOT NULL;
+-----------+-----------------+
| THREAD_ID | NUMBER_OF_BYTES |
+-----------+-----------------+
|        11 |              66 |
|        11 |              47 |
|        11 |             139 |
|         5 |              24 |
|         5 |             834 |
+-----------+-----------------+
```

Most Performance Schema tables have indexes, which gives the optimizer access to execution plans other than full table scans. These indexes also improve performance for related objects, such as `sys` schema views that use those tables. For more information, see Optimizing Performance Schema Queries.

Chapter 7 Performance Schema Instrument Naming Conventions

An instrument name consists of a sequence of elements separated by `'/'` characters. Example names:

```
wait/io/file/myisam/log
wait/io/file/mysys/charset
wait/lock/table/sql/handler
wait/synch/cond/mysys/COND_alarm
wait/synch/cond/sql/BINLOG::update_cond
wait/synch/mutex/mysys/BITMAP_mutex
wait/synch/mutex/sql/LOCK_delete
wait/synch/rwlock/sql/Query_cache_query::lock
stage/sql/closing tables
stage/sql/Sorting result
statement/com/Execute
statement/com/Query
statement/sql/create_table
statement/sql/lock_tables
errors
```

The instrument name space has a tree-like structure. The elements of an instrument name from left to right provide a progression from more general to more specific. The number of elements a name has depends on the type of instrument.

The interpretation of a given element in a name depends on the elements to the left of it. For example, `myisam` appears in both of the following names, but `myisam` in the first name is related to file I/O, whereas in the second it is related to a synchronization instrument:

```
wait/io/file/myisam/log
wait/synch/cond/myisam/MI_SORT_INFO::cond
```

Instrument names consist of a prefix with a structure defined by the Performance Schema implementation and a suffix defined by the developer implementing the instrument code. The top-level element of an instrument prefix indicates the type of instrument. This element also determines which event timer in the `performance_timers` table applies to the instrument. For the prefix part of instrument names, the top level indicates the type of instrument.

The suffix part of instrument names comes from the code for the instruments themselves. Suffixes may include levels such as these:

- A name for the major element (a server module such as `myisam`, `innodb`, `mysys`, or `sql`) or a plugin name.
- The name of a variable in the code, in the form *`XXX`* (a global variable) or *`CCC`*`::`*`MMM`* (a member *`MMM`* in class *`CCC`*). Examples: `COND_thread_cache`, `THR_LOCK_myisam`, `BINLOG::LOCK_index`.

- Wait Instrument Elements

Top-Level Instrument Elements

- `idle`: An instrumented idle event. This instrument has no further elements.
- `error`: An instrumented error event. This instrument has no further elements.
- `memory`: An instrumented memory event.
- `stage`: An instrumented stage event.
- `statement`: An instrumented statement event.
- `transaction`: An instrumented transaction event. This instrument has no further elements.
- `wait`: An instrumented wait event.

Idle Instrument Elements

The `idle` instrument is used for idle events, which The Performance Schema generates as discussed in the description of the `socket_instances.STATE` column in Section 10.3.5, “The socket_instances Table”.

Error Instrument Elements

The `error` instrument indicates whether to collect information for server errors and warnings. This instrument is enabled by default. The `TIMED` column for the `error` row in the `setup_instruments` table is inapplicable because timing information is not collected.

Memory Instrument Elements

Memory instrumentation is enabled by default. Memory instrumentation can be enabled or disabled at startup, or dynamically at runtime by updating the `ENABLED` column of the relevant instruments in the `setup_instruments` table. Memory instruments have names of the form `memory/`*`code_area`*`/`*`instrument_name`* where *`code_area`* is a value such as `sql` or `myisam`, and *`instrument_name`* is the instrument detail.

Instruments named with the prefix `memory/performance_schema/` expose how much memory is allocated for internal buffers in the Performance Schema. The `memory/performance_schema/` instruments are built in, always enabled, and cannot be disabled at startup or runtime. Built-in memory instruments are displayed only in the `memory_summary_global_by_event_name` table. For more information, see The Performance Schema Memory-Allocation Model.

Stage Instrument Elements

Stage instruments have names of the form `stage/`*`code_area`*`/`*`stage_name`*, where *`code_area`* is a value such as `sql` or `myisam`, and *`stage_name`* indicates the stage of statement processing, such as `Sorting result` or `Sending data`. Stages correspond to the thread states displayed by `SHOW PROCESSLIST` or that are visible in the Information Schema `PROCESSLIST` table.

Statement Instrument Elements

- `statement/abstract/*`: An abstract instrument for statement operations. Abstract instruments are used during the early stages of statement classification before the exact statement type is known,

then changed to a more specific statement instrument when the type is known. For a description of this process, see Section 10.6, “Performance Schema Statement Event Tables”.

- `statement/com`: An instrumented command operation. These have names corresponding to `COM_xxx` operations (see the `mysql_com.h` header file and `sql/sql_parse.cc`. For example, the `statement/com/Connect` and `statement/com/Init DB` instruments correspond to the `COM_CONNECT` and `COM_INIT_DB` commands.
- `statement/scheduler/event`: A single instrument to track all events executed by the Event Scheduler. This instrument comes into play when a scheduled event begins executing.
- `statement/sp`: An instrumented internal instruction executed by a stored program. For example, the `statement/sp/cfetch` and `statement/sp/freturn` instruments are used cursor fetch and function return instructions.
- `statement/sql`: An instrumented SQL statement operation. For example, the `statement/sql/create_db` and `statement/sql/select` instruments are used for `CREATE DATABASE` and `SELECT` statements.

Thread Instrument Elements

Instrumented threads are displayed in the `setup_threads` table, which exposes thread class names and attributes.

Thread instruments begin with `thread` (for example, `thread/sql/parser_service` or `thread/performance_schema/setup`).

The names of thread instruments for `ndbcluster` plugin threads begin with `thread/ndbcluster/`; for more information about these, see ndbcluster Plugin Threads.

Wait Instrument Elements

- `wait/io`

  An instrumented I/O operation.

  - `wait/io/file`

    An instrumented file I/O operation. For files, the wait is the time waiting for the file operation to complete (for example, a call to `fwrite()`). Due to caching, the physical file I/O on the disk might not happen within this call.

  - `wait/io/socket`

    An instrumented socket operation. Socket instruments have names of the form `wait/io/socket/sql/socket_type`. The server has a listening socket for each network protocol that it supports. The instruments associated with listening sockets for TCP/IP or Unix socket file connections have a *`socket_type`* value of `server_tcpip_socket` or `server_unix_socket`, respectively. When a listening socket detects a connection, the server transfers the connection to a new socket managed by a separate thread. The instrument for the new connection thread has a *`socket_type`* value of `client_connection`.

  - `wait/io/table`

    An instrumented table I/O operation. These include row-level accesses to persistent base tables or temporary tables. Operations that affect rows are fetch, insert, update, and delete. For a view, waits are associated with base tables referenced by the view.

Unlike most waits, a table I/O wait can include other waits. For example, table I/O might include file I/O or memory operations. Thus, `events_waits_current` for a table I/O wait usually has two rows. For more information, see Performance Schema Atom and Molecule Events.

Some row operations might cause multiple table I/O waits. For example, an insert might activate a trigger that causes an update.

- `wait/lock`

  An instrumented lock operation.

  - `wait/lock/table`

    An instrumented table lock operation.

  - `wait/lock/metadata/sql/mdl`

    An instrumented metadata lock operation.

- `wait/synch`

  An instrumented synchronization object. For synchronization objects, the `TIMER_WAIT` time includes the amount of time blocked while attempting to acquire a lock on the object, if any.

  - `wait/synch/cond`

    A condition is used by one thread to signal to other threads that something they were waiting for has happened. If a single thread was waiting for a condition, it can wake up and proceed with its execution. If several threads were waiting, they can all wake up and compete for the resource for which they were waiting.

  - `wait/synch/mutex`

    A mutual exclusion object used to permit access to a resource (such as a section of executable code) while preventing other threads from accessing the resource.

  - `wait/synch/prlock`

    A priority rwlock lock object.

  - `wait/synch/rwlock`

    A plain read/write lock object used to lock a specific variable for access while preventing its use by other threads. A shared read lock can be acquired simultaneously by multiple threads. An exclusive write lock can be acquired by only one thread at a time.

  - `wait/synch/sxlock`

    A shared-exclusive (SX) lock is a type of rwlock lock object that provides write access to a common resource while permitting inconsistent reads by other threads. `sxlocks` optimize concurrency and improve scalability for read-write workloads.

Chapter 8 Performance Schema Status Monitoring

There are several status variables associated with the Performance Schema:

```
mysql> SHOW STATUS LIKE 'perf%';
+-------------------------------------------+-------+
| Variable_name                             | Value |
+-------------------------------------------+-------+
| Performance_schema_accounts_lost          | 0     |
| Performance_schema_cond_classes_lost      | 0     |
| Performance_schema_cond_instances_lost    | 0     |
| Performance_schema_digest_lost            | 0     |
| Performance_schema_file_classes_lost      | 0     |
| Performance_schema_file_handles_lost      | 0     |
| Performance_schema_file_instances_lost    | 0     |
| Performance_schema_hosts_lost             | 0     |
| Performance_schema_locker_lost            | 0     |
| Performance_schema_memory_classes_lost    | 0     |
| Performance_schema_metadata_lock_lost     | 0     |
| Performance_schema_mutex_classes_lost     | 0     |
| Performance_schema_mutex_instances_lost   | 0     |
| Performance_schema_nested_statement_lost  | 0     |
| Performance_schema_program_lost           | 0     |
| Performance_schema_rwlock_classes_lost    | 0     |
| Performance_schema_rwlock_instances_lost  | 0     |
| Performance_schema_session_connect_attrs_lost | 0     |
| Performance_schema_socket_classes_lost    | 0     |
| Performance_schema_socket_instances_lost  | 0     |
| Performance_schema_stage_classes_lost     | 0     |
| Performance_schema_statement_classes_lost | 0     |
| Performance_schema_table_handles_lost     | 0     |
| Performance_schema_table_instances_lost   | 0     |
| Performance_schema_thread_classes_lost    | 0     |
| Performance_schema_thread_instances_lost  | 0     |
| Performance_schema_users_lost             | 0     |
+-------------------------------------------+-------+
```

The Performance Schema status variables provide information about instrumentation that could not be loaded or created due to memory constraints. Names for these variables have several forms:

- `Performance_schema_xxx_classes_lost` indicates how many instruments of type *`xxx`* could not be loaded.
- `Performance_schema_xxx_instances_lost` indicates how many instances of object type *`xxx`* could not be created.
- `Performance_schema_xxx_handles_lost` indicates how many instances of object type *`xxx`* could not be opened.
- `Performance_schema_locker_lost` indicates how many events are “lost” or not recorded.

For example, if a mutex is instrumented in the server source but the server cannot allocate memory for the instrumentation at runtime, it increments `Performance_schema_mutex_classes_lost`. The mutex still functions as a synchronization object (that is, the server continues to function normally), but performance data for it is not collected. If the instrument can be allocated, it can be used for initializing instrumented mutex instances. For a singleton mutex such as a global mutex, there is only one instance. Other mutexes have an instance per connection, or per page in various caches and data buffers, so the number of instances varies over time. Increasing the maximum number of connections or the maximum size of some buffers increases the maximum number of instances that might be allocated at once. If the server cannot create a given instrumented mutex instance, it increments `Performance_schema_mutex_instances_lost`.

Suppose that the following conditions hold:

- The server was started with the `--performance_schema_max_mutex_classes=200` option and thus has room for 200 mutex instruments.
- 150 mutex instruments have been loaded already.
- The plugin named `plugin_a` contains 40 mutex instruments.
- The plugin named `plugin_b` contains 20 mutex instruments.

The server allocates mutex instruments for the plugins depending on how many they need and how many are available, as illustrated by the following sequence of statements:

```
INSTALL PLUGIN plugin_a
```

The server now has 150+40 = 190 mutex instruments.

```
UNINSTALL PLUGIN plugin_a;
```

The server still has 190 instruments. All the historical data generated by the plugin code is still available, but new events for the instruments are not collected.

```
INSTALL PLUGIN plugin_a;
```

The server detects that the 40 instruments are already defined, so no new instruments are created, and previously assigned internal memory buffers are reused. The server still has 190 instruments.

```
INSTALL PLUGIN plugin_b;
```

The server has room for 200-190 = 10 instruments (in this case, mutex classes), and sees that the plugin contains 20 new instruments. 10 instruments are loaded, and 10 are discarded or “lost.” The `Performance_schema_mutex_classes_lost` indicates the number of instruments (mutex classes) lost:

```
mysql> SHOW STATUS LIKE "perf%mutex_classes_lost";
+---------------------------------------+-------+
| Variable_name                         | Value |
+---------------------------------------+-------+
| Performance_schema_mutex_classes_lost | 10    |
+---------------------------------------+-------+
1 row in set (0.10 sec)
```

The instrumentation still works and collects (partial) data for `plugin_b`.

When the server cannot create a mutex instrument, these results occur:

- No row for the instrument is inserted into the `setup_instruments` table.
- `Performance_schema_mutex_classes_lost` increases by 1.
- `Performance_schema_mutex_instances_lost` does not change. (When the mutex instrument is not created, it cannot be used to create instrumented mutex instances later.)

The pattern just described applies to all types of instruments, not just mutexes.

A value of `Performance_schema_mutex_classes_lost` greater than 0 can happen in two cases:

- To save a few bytes of memory, you start the server with `--performance_schema_max_mutex_classes=N`, where *N* is less than the default value. The default

value is chosen to be sufficient to load all the plugins provided in the MySQL distribution, but this can be reduced if some plugins are never loaded. For example, you might choose not to load some of the storage engines in the distribution.

- You load a third-party plugin that is instrumented for the Performance Schema but do not allow for the plugin's instrumentation memory requirements when you start the server. Because it comes from a third party, the instrument memory consumption of this engine is not accounted for in the default value chosen for `performance_schema_max_mutex_classes`.

  If the server has insufficient resources for the plugin's instruments and you do not explicitly allocate more using `--performance_schema_max_mutex_classes=N`, loading the plugin leads to starvation of instruments.

If the value chosen for `performance_schema_max_mutex_classes` is too small, no error is reported in the error log and there is no failure at runtime. However, the content of the tables in the `performance_schema` database misses events. The `Performance_schema_mutex_classes_lost` status variable is the only visible sign to indicate that some events were dropped internally due to failure to create instruments.

If an instrument is not lost, it is known to the Performance Schema, and is used when instrumenting instances. For example, `wait/synch/mutex/sql/LOCK_delete` is the name of a mutex instrument in the `setup_instruments` table. This single instrument is used when creating a mutex in the code (in `THD::LOCK_delete`) however many instances of the mutex are needed as the server runs. In this case, `LOCK_delete` is a mutex that is per connection (`THD`), so if a server has 1000 connections, there are 1000 threads, and 1000 instrumented `LOCK_delete` mutex instances (`THD::LOCK_delete`).

If the server does not have room for all these 1000 instrumented mutexes (instances), some mutexes are created with instrumentation, and some are created without instrumentation. If the server can create only 800 instances, 200 instances are lost. The server continues to run, but increments `Performance_schema_mutex_instances_lost` by 200 to indicate that instances could not be created.

A value of `Performance_schema_mutex_instances_lost` greater than 0 can happen when the code initializes more mutexes at runtime than were allocated for `--performance_schema_max_mutex_instances=N`.

The bottom line is that if `SHOW STATUS LIKE 'perf%'` says that nothing was lost (all values are zero), the Performance Schema data is accurate and can be relied upon. If something was lost, the data is incomplete, and the Performance Schema could not record everything given the insufficient amount of memory it was given to use. In this case, the specific `Performance_schema_xxx_lost` variable indicates the problem area.

It might be appropriate in some cases to cause deliberate instrument starvation. For example, if you do not care about performance data for file I/O, you can start the server with all Performance Schema parameters related to file I/O set to 0. No memory is allocated for file-related classes, instances, or handles, and all file events are lost.

Use `SHOW ENGINE PERFORMANCE_SCHEMA STATUS` to inspect the internal operation of the Performance Schema code:

```
mysql> SHOW ENGINE PERFORMANCE_SCHEMA STATUS\G
...
*************************** 3. row ***************************
  Type: performance_schema
  Name: events_waits_history.size
Status: 76
*************************** 4. row ***************************
```

```
  Type: performance_schema
  Name: events_waits_history.count
Status: 10000
*************************** 5. row ***************************
  Type: performance_schema
  Name: events_waits_history.memory
Status: 760000
...
*************************** 57. row ***************************
  Type: performance_schema
  Name: performance_schema.memory
Status: 26459600
...
```

This statement is intended to help the DBA understand the effects that different Performance Schema options have on memory requirements. For a description of the field meanings, see SHOW ENGINE Statement.

Chapter 9 Performance Schema General Table Characteristics

The name of the `performance_schema` database is lowercase, as are the names of tables within it. Queries should specify the names in lowercase.

Many tables in the `performance_schema` database are read only and cannot be modified:

```
mysql> TRUNCATE TABLE performance_schema.setup_instruments;
ERROR 1683 (HY000): Invalid performance_schema usage.
```

Some of the setup tables have columns that can be modified to affect Performance Schema operation; some also permit rows to be inserted or deleted. Truncation is permitted to clear collected events, so `TRUNCATE TABLE` can be used on tables containing those kinds of information, such as tables named with a prefix of `events_waits_`.

Summary tables can be truncated with `TRUNCATE TABLE`. 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.

Privileges are as for other databases and tables:

- To retrieve from `performance_schema` tables, you must have the `SELECT` privilege.
- To change those columns that can be modified, you must have the `UPDATE` privilege.
- To truncate tables that can be truncated, you must have the `DROP` privilege.

Because only a limited set of privileges apply to Performance Schema tables, attempts to use `GRANT ALL` as shorthand for granting privileges at the database or table level fail with an error:

```
mysql> GRANT ALL ON performance_schema.*
       TO 'u1'@'localhost';
ERROR 1044 (42000): Access denied for user 'root'@'localhost'
to database 'performance_schema'
mysql> GRANT ALL ON performance_schema.setup_instruments
       TO 'u2'@'localhost';
ERROR 1044 (42000): Access denied for user 'root'@'localhost'
to database 'performance_schema'
```

Instead, grant exactly the desired privileges:

```
mysql> GRANT SELECT ON performance_schema.*
       TO 'u1'@'localhost';
Query OK, 0 rows affected (0.03 sec)
mysql> GRANT SELECT, UPDATE ON performance_schema.setup_instruments
       TO 'u2'@'localhost';
Query OK, 0 rows affected (0.02 sec)
```

Chapter 10 Performance Schema Table Descriptions

Table of Contents

Tables in the `performance_schema` database can be grouped as follows:

- Setup tables. These tables are used to configure and display monitoring characteristics.
- Current events tables. The `events_waits_current` table contains the most recent event for each thread. Other similar tables contain current events at different levels of the event hierarchy: `events_stages_current` for stage events, `events_statements_current` for statement events, and `events_transactions_current` for transaction events.
- History tables. These tables have the same structure as the current events tables, but contain more rows. For example, for wait events, `events_waits_history` table contains the most recent 10 events per thread. `events_waits_history_long` contains the most recent 10,000 events. Other similar tables exist for stage, statement, and transaction histories.

  To change the sizes of the history tables, set the appropriate system variables at server startup. For example, to set the sizes of the wait event history tables, set `performance_schema_events_waits_history_size` and `performance_schema_events_waits_history_long_size`.
- Summary tables. These tables contain information aggregated over groups of events, including those that have been discarded from the history tables.
- Instance tables. These 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.
- Miscellaneous tables. These do not fall into any of the other table groups.

10.1 Performance Schema Table Reference

The following table summarizes all available Performance Schema tables. For greater detail, see the individual table descriptions.

Table 10.1 Performance Schema Tables

Table Name	Description	Introduced
`accounts`	Connection statistics per client account	
`binary_log_transaction_compression_stats`	Binary log transaction compression	8.0.20
`clone_progress`	Clone operation progress	8.0.17
`clone_status`	Clone operation status	8.0.17
`component_scheduler_tasks`	Status of scheduled tasks	8.0.34
`cond_instances`	Synchronization object instances	
`data_lock_waits`	Data lock wait relationships	
`data_locks`	Data locks held and requested	
`error_log`	Server error log recent entries	8.0.22
`events_errors_summary_by_account_by_error`	Errors per account and error code	
`events_errors_summary_by_host_by_error`	Errors per host and error code	

Table Name	Description	Introduced
events_errors_summary_by_thread_by_error	Errors per thread and error code	
events_errors_summary_by_user_by_error	Errors per user and error code	
events_errors_summary_global_by_error	Errors per error code	
events_stages_current	Current stage events	
events_stages_history	Most recent stage events per thread	
events_stages_history_long	Most recent stage events overall	
events_stages_summary_by_account_by_event_name	Stage events per account and event name	
events_stages_summary_by_host_by_event_name	Stage events per host name and event name	
events_stages_summary_by_thread_by_event_name	Stage waits per thread and event name	
events_stages_summary_by_user_by_event_name	Stage events per user name and event name	
events_stages_summary_global_by_event_name	Stage waits per event name	
events_statements_current	Current statement events	
events_statements_histogram_by_digest	Statement histograms per schema and digest value	
events_statements_histogram_global	Statement histogram summarized globally	
events_statements_history	Most recent statement events per thread	
events_statements_history_long	Most recent statement events overall	
events_statements_summary_by_account_by_event_name	Statement events per account and event name	
events_statements_summary_by_digest	Statement events per schema and digest value	
events_statements_summary_by_host_by_event_name	Statement events per host name and event name	
events_statements_summary_by_program	Statement events per stored program	
events_statements_summary_by_thread_by_event_name	Statement events per thread and event name	
events_statements_summary_by_user_by_event_name	Statement events per user name and event name	
events_statements_summary_global_by_event_name	Statement events per event name	
events_transactions_current	Current transaction events	
events_transactions_history	Most recent transaction events per thread	

Table Name	Description	Introduced
events_transactions_history_long	Most recent transaction events overall	
events_transactions_summary_by_account_by_event_name	Transaction events per account and event name	
events_transactions_summary_by_host_by_event_name	Transaction events per host name and event name	
events_transactions_summary_by_thread_by_event_name	Transaction events per thread and event name	
events_transactions_summary_by_user_by_event_name	Transaction events per user name and event name	
events_transactions_summary_global_by_event_name	Transaction events per event name	
events_waits_current	Current wait events	
events_waits_history	Most recent wait events per thread	
events_waits_history_long	Most recent wait events overall	
events_waits_summary_by_account_by_event_name	Wait events per account and event name	
events_waits_summary_by_host_by_event_name	Wait events per host name and event name	
events_waits_summary_by_instance	Wait events per instance	
events_waits_summary_by_thread_by_event_name	Wait events per thread and event name	
events_waits_summary_by_user_by_event_name	Wait events per user name and event name	
events_waits_summary_global_by_event_name	Wait events per event name	
file_instances	File instances	
file_summary_by_event_name	File events per event name	
file_summary_by_instance	File events per file instance	
firewall_group_allowlist	Firewall in-memory data for group profile allowlists	8.0.23
firewall_groups	Firewall in-memory data for group profiles	8.0.23
firewall_membership	Firewall in-memory data for group profile members	8.0.23
global_status	Global status variables	
global_variables	Global system variables	
host_cache	Information from internal host cache	
hosts	Connection statistics per client host name	
keyring_component_status	Status information for installed keyring component	8.0.24

Table Name	Description	Introduced
keyring_keys	Metadata for keyring keys	8.0.16
log_status	Information about server logs for backup purposes	
memory_summary_by_account_by_event_name	Memory operations per account and event name	
memory_summary_by_host_by_event_name	Memory operations per host and event name	
memory_summary_by_thread_by_event_name	Memory operations per thread and event name	
memory_summary_by_user_by_event_name	Memory operations per user and event name	
memory_summary_global_by_event_name	Memory operations globally per event name	
metadata_locks	Metadata locks and lock requests	
mutex_instances	Mutex synchronization object instances	
ndb_sync_excluded_objects	NDB objects which cannot be synchronized	8.0.21
ndb_sync_pending_objects	NDB objects waiting for synchronization	8.0.21
objects_summary_global_by_type	Object summaries	
performance_timers	Which event timers are available	
persisted_variables	Contents of mysqld-auto.cnf file	
prepared_statements_instances	Prepared statement instances and statistics	
processlist	Process list information	8.0.22
replication_applier_configuration	Configuration parameters for replication applier on replica	
replication_applier_filters	Channel-specific replication filters on current replica	
replication_applier_global_filters	Global replication filters on current replica	
replication_applier_status	Current status of replication applier on replica	
replication_applier_status_by_coordinator	SQL or coordinator thread applier status	
replication_applier_status_by_worker	Worker thread applier status	
replication_asynchronous_connection_failover	Source lists for asynchronous connection failover mechanism	8.0.22
replication_asynchronous_connection_failover_managed	Managed source lists for asynchronous connection failover mechanism	8.0.23

Table Name	Description	Introduced
replication_connection_configuration	Configuration parameters for connecting to source	
replication_connection_status	Current status of connection to source	
replication_group_communication_information	Replication group configuration options	8.0.27
replication_group_member_stats	Replication group member statistics	
replication_group_members	Replication group member network and status	
rwlock_instances	Lock synchronization object instances	
session_account_connect_attrs	Connection attributes per for current session	
session_connect_attrs	Connection attributes for all sessions	
session_status	Status variables for current session	
session_variables	System variables for current session	
setup_actors	How to initialize monitoring for new foreground threads	
setup_consumers	Consumers for which event information can be stored	
setup_instruments	Classes of instrumented objects for which events can be collected	
setup_objects	Which objects should be monitored	
setup_threads	Instrumented thread names and attributes	
socket_instances	Active connection instances	
socket_summary_by_event_name	Socket waits and I/O per event name	
socket_summary_by_instance	Socket waits and I/O per instance	
status_by_account	Session status variables per account	
status_by_host	Session status variables per host name	
status_by_thread	Session status variables per session	
status_by_user	Session status variables per user name	
table_handles	Table locks and lock requests	

Table Name	Description	Introduced
table_io_waits_summary_by_index_usage	Table I/O waits per index	
table_io_waits_summary_by_table	Table I/O waits per table	
table_lock_waits_summary_by_table	Table lock waits per table	
threads	Information about server threads	
tls_channel_status	TLS status for each connection interface	8.0.21
tp_thread_group_state	Thread pool thread group states	8.0.14
tp_thread_group_stats	Thread pool thread group statistics	8.0.14
tp_thread_state	Thread pool thread information	8.0.14
user_defined_functions	Registered loadable functions	
user_variables_by_thread	User-defined variables per thread	
users	Connection statistics per client user name	
variables_by_thread	Session system variables per session	
variables_info	How system variables were most recently set	

10.2 Performance Schema Setup Tables

The setup tables provide information about the current instrumentation and enable the monitoring configuration to be changed. For this reason, some columns in these tables can be changed if you have the UPDATE privilege.

The use of tables rather than individual variables for setup information provides a high degree of flexibility in modifying Performance Schema configuration. For example, you can use a single statement with standard SQL syntax to make multiple simultaneous configuration changes.

These setup tables are available:

- setup_actors: How to initialize monitoring for new foreground threads
- setup_consumers: The destinations to which event information can be sent and stored
- setup_instruments: The classes of instrumented objects for which events can be collected
- setup_objects: Which objects should be monitored
- setup_threads: Instrumented thread names and attributes

10.2.1 The setup_actors Table

The setup_actors table contains information that determines whether to enable monitoring and historical event logging for new foreground server threads (threads associated with client connections). This table has a maximum size of 100 rows by default. To change the table size, modify the performance_schema_setup_actors_size system variable at server startup.

For each new foreground thread, the Performance Schema matches the user and host for the thread against the rows of the setup_actors table. If a row from that table matches, its ENABLED and HISTORY

column values are used to set the INSTRUMENTED and HISTORY columns, respectively, of the threads table row for the thread. This enables instrumenting and historical event logging to be applied selectively per host, user, or account (user and host combination). If there is no match, the INSTRUMENTED and HISTORY columns for the thread are set to NO.

For background threads, there is no associated user. INSTRUMENTED and HISTORY are YES by default and setup_actors is not consulted.

The initial contents of the setup_actors table match any user and host combination, so monitoring and historical event collection are enabled by default for all foreground threads:

```
mysql> SELECT * FROM performance_schema.setup_actors;
+------+------+------+---------+---------+
| HOST | USER | ROLE | ENABLED | HISTORY |
+------+------+------+---------+---------+
| %    | %    | %    | YES     | YES     |
+------+------+------+---------+---------+
```

For information about how to use the setup_actors table to affect event monitoring, see Section 5.6, “Pre-Filtering by Thread”.

Modifications to the setup_actors table affect only foreground threads created subsequent to the modification, not existing threads. To affect existing threads, modify the INSTRUMENTED and HISTORY columns of threads table rows.

The setup_actors table has these columns:

- HOST

  The host name. This should be a literal name, or '%' to mean “any host.”

- USER

  The user name. This should be a literal name, or '%' to mean “any user.”

- ROLE

  Unused.

- ENABLED

  Whether to enable instrumentation for foreground threads matched by the row. The value is YES or NO.

- HISTORY

  Whether to log historical events for foreground threads matched by the row. The value is YES or NO.

The setup_actors table has these indexes:

- Primary key on (HOST, USER, ROLE)

TRUNCATE TABLE is permitted for the setup_actors table. It removes the rows.

10.2.2 The setup_consumers Table

The setup_consumers table lists the types of consumers for which event information can be stored and which 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_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     |
+----------------------------------+---------+
```

The consumer settings in the `setup_consumers` table form a hierarchy from higher levels to lower. For detailed information about the effect of enabling different consumers, see Section 5.7, “Pre-Filtering by Consumer”.

Modifications to the `setup_consumers` table affect monitoring immediately.

The `setup_consumers` table has these columns:

- `NAME`

  The consumer name.

- `ENABLED`

  Whether the consumer is enabled. The value is `YES` or `NO`. This column can be modified. If you disable a consumer, the server does not spend time adding event information to it.

The `setup_consumers` table has these indexes:

- Primary key on (`NAME`)

`TRUNCATE TABLE` is not permitted for the `setup_consumers` table.

10.2.3 The setup_instruments Table

The `setup_instruments` table lists classes of instrumented objects for which events can be collected:

```
mysql> SELECT * FROM performance_schema.setup_instruments\G
*************************** 1. row ***************************
         NAME: wait/synch/mutex/pfs/LOCK_pfs_share_list
      ENABLED: NO
        TIMED: NO
   PROPERTIES: singleton
        FLAGS: NULL
   VOLATILITY: 1
DOCUMENTATION: Components can provide their own performance_schema tables.
This lock protects the list of such tables definitions.
...
*************************** 410. row ***************************
         NAME: stage/sql/executing
      ENABLED: NO
        TIMED: NO
   PROPERTIES:
        FLAGS: NULL
   VOLATILITY: 0
DOCUMENTATION: NULL
...
```

```
*************************** 733. row ***************************
         NAME: statement/abstract/Query
      ENABLED: YES
        TIMED: YES
   PROPERTIES: mutable
        FLAGS: NULL
   VOLATILITY: 0
DOCUMENTATION: SQL query just received from the network.
At this point, the real statement type is unknown, the type
will be refined after SQL parsing.
...
*************************** 737. row ***************************
         NAME: memory/performance_schema/mutex_instances
      ENABLED: YES
        TIMED: NULL
   PROPERTIES: global_statistics
        FLAGS:
   VOLATILITY: 1
DOCUMENTATION: Memory used for table performance_schema.mutex_instances
...
*************************** 823. row ***************************
         NAME: memory/sql/Prepared_statement::infrastructure
      ENABLED: YES
        TIMED: NULL
   PROPERTIES: controlled_by_default
        FLAGS: controlled
   VOLATILITY: 0
DOCUMENTATION: Map infrastructure for prepared statements per session.
...
```

Each instrument added to the source code provides a row for the `setup_instruments` table, even when the instrumented code is not executed. When an instrument is enabled and executed, instrumented instances are created, which are visible in the *`xxx`*`_instances` tables, such as `file_instances` or `rwlock_instances`.

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.

For more information about the role of the `setup_instruments` table in event filtering, see Section 5.3, “Event Pre-Filtering”.

The `setup_instruments` table has these columns:

- `NAME`

  The instrument name. Instrument names may have multiple parts and form a hierarchy, as discussed in Chapter 7, *Performance Schema Instrument Naming Conventions*. Events produced from execution of an instrument have an `EVENT_NAME` value that is taken from the instrument `NAME` value. (Events do not really have a “name,” but this provides a way to associate events with instruments.)

- `ENABLED`

  Whether the instrument is enabled. The value is `YES` or `NO`. A disabled instrument produces no events. This column can be modified, although setting `ENABLED` has no effect for instruments that have already been created.

- `TIMED`

  Whether the instrument is timed. The value is `YES`, `NO`, or `NULL`. This column can be modified, although setting `TIMED` has no effect for instruments that have already been created.

A `TIMED` value of `NULL` indicates that the instrument does not support timing. For example, memory operations are not timed, so their `TIMED` column is `NULL`.

Setting `TIMED` to `NULL` for an instrument that supports timing has no effect, as does setting `TIMED` to non-`NULL` for an instrument that does not support timing.

If an enabled instrument is not timed, the instrument code is enabled, but the timer is not. 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 the sum, minimum, maximum, and average time values in summary tables.

- `PROPERTIES`

  The instrument properties. This column uses the `SET` data type, so multiple flags from the following list can be set per instrument:

  - `controlled_by_default`: memory is collected by default for this instrument.
  - `global_statistics`: The instrument produces only global summaries. Summaries for finer levels are unavailable, such as per thread, account, user, or host. For example, most memory instruments produce only global summaries.
  - `mutable`: The instrument can “mutate” into a more specific one. This property applies only to statement instruments.
  - `progress`: The instrument is capable of reporting progress data. This property applies only to stage instruments.
  - `singleton`: The instrument has a single instance. For example, most global mutex locks in the server are singletons, so the corresponding instruments are as well.
  - `user`: The instrument is directly related to user workload (as opposed to system workload). One such instrument is `wait/io/socket/sql/client_connection`.

- `FLAGS`

  Whether the instrument's memory is controlled.

  This flag is supported for non-global memory instruments, only, and can be set or unset. For example:

  ```
  SQL> UPDATE PERFORMANCE_SCHEMA.SETUP_INTRUMENTS SET FLAGS="controlled" WHERE NAME='memory/sql/
  ```

  > **Note**
  >
  > Attempting to set `FLAGS = controlled` on non-memory instruments, or on global memory instruments, fails silently.

- `VOLATILITY`

  The instrument volatility. Volatility values range from low to high. The values correspond to the `PSI_VOLATILITY_`*`xxx`* constants defined in the `mysql/psi/psi_base.h` header file:

  ```
  #define PSI_VOLATILITY_UNKNOWN 0
  #define PSI_VOLATILITY_PERMANENT 1
  #define PSI_VOLATILITY_PROVISIONING 2
  #define PSI_VOLATILITY_DDL 3
  ```

```
#define PSI_VOLATILITY_CACHE 4
#define PSI_VOLATILITY_SESSION 5
#define PSI_VOLATILITY_TRANSACTION 6
#define PSI_VOLATILITY_QUERY 7
#define PSI_VOLATILITY_INTRA_QUERY 8
```

The `VOLATILITY` column is purely informational, to provide users (and the Performance Schema code) some hint about the instrument runtime behavior.

Instruments with a low volatility index (PERMANENT = 1) are created once at server startup, and never destroyed or re-created during normal server operation. They are destroyed only during server shutdown.

For example, the `wait/synch/mutex/pfs/LOCK_pfs_share_list` mutex is defined with a volatility of 1, which means it is created once. Possible overhead from the instrumentation itself (namely, mutex initialization) has no effect for this instrument then. Runtime overhead occurs only when locking or unlocking the mutex.

Instruments with a higher volatility index (for example, SESSION = 5) are created and destroyed for every user session. For example, the `wait/synch/mutex/sql/THD::LOCK_query_plan` mutex is created each time a session connects, and destroyed when the session disconnects.

This mutex is more sensitive to Performance Schema overhead, because overhead comes not only from the lock and unlock instrumentation, but also from mutex create and destroy instrumentation, which is executed more often.

Another aspect of volatility concerns whether and when an update to the `ENABLED` column actually has some effect:

- An update to `ENABLED` affects instrumented objects created subsequently, but has no effect on instruments already created.
- Instruments that are more “volatile” use new settings from the `setup_instruments` table sooner.

For example, this statement does not affect the `LOCK_query_plan` mutex for existing sessions, but does have an effect on new sessions created subsequent to the update:

```
UPDATE performance_schema.setup_instruments
SET ENABLED=value
WHERE NAME = 'wait/synch/mutex/sql/THD::LOCK_query_plan';
```

This statement actually has no effect at all:

```
UPDATE performance_schema.setup_instruments
SET ENABLED=value
WHERE NAME = 'wait/synch/mutex/pfs/LOCK_pfs_share_list';
```

This mutex is permanent, and was created already before the update is executed. The mutex is never created again, so the `ENABLED` value in `setup_instruments` is never used. To enable or disable this mutex, use the `mutex_instances` table instead.

- `DOCUMENTATION`

  A string describing the instrument purpose. The value is `NULL` if no description is available.

The `setup_instruments` table has these indexes:

- Primary key on (`NAME`)

`TRUNCATE TABLE` is not permitted for the `setup_instruments` table.

As of MySQL 8.0.27, to assist monitoring and troubleshooting, the Performance Schema instrumentation is used to export names of instrumented threads to the operating system. This enables utilities that display thread names, such as debuggers and the Unix `ps` command, to display distinct `mysqld` thread names rather than “mysqld”. This feature is supported only on Linux, macOS, and Windows.

Suppose that `mysqld` is running on a system that has a version of `ps` that supports this invocation syntax:

```
ps -C mysqld H -o "pid tid cmd comm"
```

Without export of thread names to the operating system, the command displays output like this, where most `COMMAND` values are `mysqld`:

```
  PID   TID CMD                         COMMAND
 1377  1377 /usr/sbin/mysqld            mysqld
 1377  1528 /usr/sbin/mysqld            mysqld
 1377  1529 /usr/sbin/mysqld            mysqld
 1377  1530 /usr/sbin/mysqld            mysqld
 1377  1531 /usr/sbin/mysqld            mysqld
 1377  1534 /usr/sbin/mysqld            mysqld
 1377  1535 /usr/sbin/mysqld            mysqld
 1377  1588 /usr/sbin/mysqld            xpl_worker1
 1377  1589 /usr/sbin/mysqld            xpl_worker0
 1377  1590 /usr/sbin/mysqld            mysqld
 1377  1594 /usr/sbin/mysqld            mysqld
 1377  1595 /usr/sbin/mysqld            mysqld
```

With export of thread names to the operating system, the output looks like this, with threads having a name similar to their instrument name:

```
  PID   TID CMD                         COMMAND
27668 27668 /usr/sbin/mysqld            mysqld
27668 27671 /usr/sbin/mysqld            ib_io_ibuf
27668 27672 /usr/sbin/mysqld            ib_io_log
27668 27673 /usr/sbin/mysqld            ib_io_rd-1
27668 27674 /usr/sbin/mysqld            ib_io_rd-2
27668 27677 /usr/sbin/mysqld            ib_io_wr-1
27668 27678 /usr/sbin/mysqld            ib_io_wr-2
27668 27699 /usr/sbin/mysqld            xpl_worker-2
27668 27700 /usr/sbin/mysqld            xpl_accept-1
27668 27710 /usr/sbin/mysqld            evt_sched
27668 27711 /usr/sbin/mysqld            sig_handler
27668 27933 /usr/sbin/mysqld            connection
```

Different thread instances within the same class are numbered to provide distinct names where that is feasible. Due to constraints on name lengths with respect to potentially large numbers of connections, connections are named simply `connection`.

10.2.4 The setup_objects Table

The `setup_objects` table controls whether the Performance Schema monitors particular objects. This table has a maximum size of 100 rows by default. To change the table size, modify the `performance_schema_setup_objects_size` system variable at server startup.

The initial `setup_objects` contents look like this:

```
mysql> SELECT * FROM performance_schema.setup_objects;
+-------------+--------------------+-------------+---------+-------+
| OBJECT_TYPE | OBJECT_SCHEMA      | OBJECT_NAME | ENABLED | TIMED |
+-------------+--------------------+-------------+---------+-------+
| EVENT       | mysql              | %           | NO      | NO    |
| EVENT       | performance_schema | %           | NO      | NO    |
```

```
| EVENT       | information_schema | %           | NO      | NO    |
| EVENT       | %                  | %           | YES     | YES   |
| FUNCTION    | mysql              | %           | NO      | NO    |
| FUNCTION    | performance_schema | %           | NO      | NO    |
| FUNCTION    | information_schema | %           | NO      | NO    |
| FUNCTION    | %                  | %           | YES     | YES   |
| PROCEDURE   | mysql              | %           | NO      | NO    |
| PROCEDURE   | performance_schema | %           | NO      | NO    |
| PROCEDURE   | information_schema | %           | NO      | NO    |
| PROCEDURE   | %                  | %           | YES     | YES   |
| TABLE       | mysql              | %           | NO      | NO    |
| TABLE       | performance_schema | %           | NO      | NO    |
| TABLE       | information_schema | %           | NO      | NO    |
| TABLE       | %                  | %           | YES     | YES   |
| TRIGGER     | mysql              | %           | NO      | NO    |
| TRIGGER     | performance_schema | %           | NO      | NO    |
| TRIGGER     | information_schema | %           | NO      | NO    |
| TRIGGER     | %                  | %           | YES     | YES   |
+-------------+--------------------+-------------+---------+-------+
```

Modifications to the `setup_objects` table affect object monitoring immediately.

For object types listed in `setup_objects`, the Performance Schema uses the table to how to monitor them. Object matching is based on the `OBJECT_SCHEMA` and `OBJECT_NAME` columns. Objects for which there is no match are not monitored.

The effect of the default object configuration is to instrument all tables 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 example, with a table `db1.t1`, it looks 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.

Rows can be inserted into or deleted from `setup_objects` by users with the `INSERT` or `DELETE` privilege on the table. For existing rows, only the `ENABLED` and `TIMED` columns can be modified, by users with the `UPDATE` privilege on the table.

For more information about the role of the `setup_objects` table in event filtering, see Section 5.3, “Event Pre-Filtering”.

The `setup_objects` table has these columns:

- `OBJECT_TYPE`

  The type of object to instrument. The value is one of `'EVENT'` (Event Scheduler event), `'FUNCTION'` (stored function), `'PROCEDURE'` (stored procedure), `'TABLE'` (base table), or `'TRIGGER'` (trigger).

  `TABLE` filtering affects table I/O events (`wait/io/table/sql/handler` instrument) and table lock events (`wait/lock/table/sql/handler` instrument).

- `OBJECT_SCHEMA`

  The schema that contains the object. This should be a literal name, or `'%'` to mean “any schema.”

- `OBJECT_NAME`

  The name of the instrumented object. This should be a literal name, or `'%'` to mean “any object.”

- ENABLED

  Whether events for the object are instrumented. The value is YES or NO. This column can be modified.

- TIMED

  Whether events for the object are timed. This column can be modified.

The setup_objects table has these indexes:

- Index on (OBJECT_TYPE, OBJECT_SCHEMA, OBJECT_NAME)

TRUNCATE TABLE is permitted for the setup_objects table. It removes the rows.

10.2.5 The setup_threads Table

The setup_threads table lists instrumented thread classes. It exposes thread class names and attributes:

```
mysql> SELECT * FROM performance_schema.setup_threads\G
*************************** 1. row ***************************
         NAME: thread/performance_schema/setup
      ENABLED: YES
      HISTORY: YES
   PROPERTIES: singleton
   VOLATILITY: 0
DOCUMENTATION: NULL
...
*************************** 4. row ***************************
         NAME: thread/sql/main
      ENABLED: YES
      HISTORY: YES
   PROPERTIES: singleton
   VOLATILITY: 0
DOCUMENTATION: NULL
*************************** 5. row ***************************
         NAME: thread/sql/one_connection
      ENABLED: YES
      HISTORY: YES
   PROPERTIES: user
   VOLATILITY: 0
DOCUMENTATION: NULL
...
*************************** 10. row ***************************
         NAME: thread/sql/event_scheduler
      ENABLED: YES
      HISTORY: YES
   PROPERTIES: singleton
   VOLATILITY: 0
DOCUMENTATION: NULL
```

The setup_threads table has these columns:

- NAME

  The instrument name. Thread instruments begin with thread (for example, thread/sql/parser_service or thread/performance_schema/setup).

- ENABLED

  Whether the instrument is enabled. The value is YES or NO. This column can be modified, although setting ENABLED has no effect for threads that are already running.

For background threads, setting the ENABLED value controls whether INSTRUMENTED is set to YES or NO for threads that are subsequently created for this instrument and listed in the threads table. For foreground threads, this column has no effect; the setup_actors table takes precedence.

- HISTORY

  Whether to log historical events for the instrument. The value is YES or NO. This column can be modified, although setting HISTORY has no effect for threads that are already running.

  For background threads, setting the HISTORY value controls whether HISTORY is set to YES or NO for threads that are subsequently created for this instrument and listed in the threads table. For foreground threads, this column has no effect; the setup_actors table takes precedence.

- PROPERTIES

  The instrument properties. This column uses the SET data type, so multiple flags from the following list can be set per instrument:

  - singleton: The instrument has a single instance. For example, there is only one thread for the thread/sql/main instrument.
  - user: The instrument is directly related to user workload (as opposed to system workload). For example, threads such as thread/sql/one_connection executing a user session have the user property to differentiate them from system threads.

- VOLATILITY

  The instrument volatility. This column has the same meaning as in the setup_instruments table. See Section 10.2.3, “The setup_instruments Table”.

- DOCUMENTATION

  A string describing the instrument purpose. The value is NULL if no description is available.

The setup_threads table has these indexes:

- Primary key on (NAME)

TRUNCATE TABLE is not permitted for the setup_threads table.

10.3 Performance Schema Instance Tables

Instance tables document what types of objects are instrumented. They provide event names and explanatory notes or status information:

- cond_instances: Condition synchronization object instances
- file_instances: File instances
- mutex_instances: Mutex synchronization object instances
- rwlock_instances: Lock synchronization object instances
- socket_instances: Active connection instances

These tables list instrumented synchronization objects, files, and connections. There are three types of synchronization objects: cond, mutex, and rwlock. Each instance table has an EVENT_NAME or NAME

column to indicate the instrument associated with each row. Instrument names may have multiple parts and form a hierarchy, as discussed in Chapter 7, *Performance Schema Instrument Naming Conventions*.

The `mutex_instances.LOCKED_BY_THREAD_ID` and `rwlock_instances.WRITE_LOCKED_BY_THREAD_ID` columns are extremely important for investigating performance bottlenecks or deadlocks. For examples of how to use them for this purpose, see Chapter 14, *Using the Performance Schema to Diagnose Problems*

10.3.1 The cond_instances Table

The `cond_instances` table lists all the conditions seen by the Performance Schema while the server executes. A condition is a synchronization mechanism used in the code to signal that a specific event has happened, so that a thread waiting for this condition can resume work.

When a thread is waiting for something to happen, the condition name is an indication of what the thread is waiting for, but there is no immediate way to tell which other thread, or threads, causes the condition to happen.

The `cond_instances` table has these columns:

- `NAME`

  The instrument name associated with the condition.

- `OBJECT_INSTANCE_BEGIN`

  The address in memory of the instrumented condition.

The `cond_instances` table has these indexes:

- Primary key on (`OBJECT_INSTANCE_BEGIN`)
- Index on (`NAME`)

`TRUNCATE TABLE` is not permitted for the `cond_instances` table.

10.3.2 The file_instances Table

The `file_instances` table lists all the files seen by the Performance Schema when executing file I/O instrumentation. If a file on disk has never been opened, it is not shown in `file_instances`. When a file is deleted from the disk, it is also removed from the `file_instances` table.

The `file_instances` table has these columns:

- `FILE_NAME`

  The file name.

- `EVENT_NAME`

  The instrument name associated with the file.

- `OPEN_COUNT`

  The count of open handles on the file. If a file was opened and then closed, it was opened 1 time, but `OPEN_COUNT` is 0. To list all the files currently opened by the server, use `WHERE OPEN_COUNT > 0`.

The `file_instances` table has these indexes:

- Primary key on (`FILE_NAME`)

- Index on (EVENT_NAME)

TRUNCATE TABLE is not permitted for the file_instances table.

10.3.3 The mutex_instances Table

The mutex_instances table lists all the mutexes seen by the Performance Schema while the server executes. A mutex is a synchronization mechanism used in the code to enforce that only one thread at a given time can have access to some common resource. The resource is said to be “protected” by the mutex.

When two threads executing in the server (for example, two user sessions executing a query simultaneously) do need to access the same resource (a file, a buffer, or some piece of data), these two threads compete against each other, so that the first query to obtain a lock on the mutex causes the other query to wait until the first is done and unlocks the mutex.

The work performed while holding a mutex is said to be in a “critical section,” and multiple queries do execute this critical section in a serialized way (one at a time), which is a potential bottleneck.

The mutex_instances table has these columns:

- NAME

  The instrument name associated with the mutex.

- OBJECT_INSTANCE_BEGIN

  The address in memory of the instrumented mutex.

- LOCKED_BY_THREAD_ID

  When a thread currently has a mutex locked, LOCKED_BY_THREAD_ID is the THREAD_ID of the locking thread, otherwise it is NULL.

The mutex_instances table has these indexes:

- Primary key on (OBJECT_INSTANCE_BEGIN)
- Index on (NAME)
- Index on (LOCKED_BY_THREAD_ID)

TRUNCATE TABLE is not permitted for the mutex_instances table.

For every mutex instrumented in the code, the Performance Schema provides the following information.

- The setup_instruments table lists the name of the instrumentation point, with the prefix wait/synch/mutex/.
- When some code creates a mutex, a row is added to the mutex_instances table. The OBJECT_INSTANCE_BEGIN column is a property that uniquely identifies the mutex.
- When a thread attempts to lock a mutex, the events_waits_current table shows a row for that thread, indicating that it is waiting on a mutex (in the EVENT_NAME column), and indicating which mutex is waited on (in the OBJECT_INSTANCE_BEGIN column).
- When a thread succeeds in locking a mutex:
  - events_waits_current shows that the wait on the mutex is completed (in the TIMER_END and TIMER_WAIT columns)

  - The completed wait event is added to the `events_waits_history` and `events_waits_history_long` tables
  - `mutex_instances` shows that the mutex is now owned by the thread (in the `THREAD_ID` column).
- When a thread unlocks a mutex, `mutex_instances` shows that the mutex now has no owner (the `THREAD_ID` column is `NULL`).
- When a mutex object is destroyed, the corresponding row is removed from `mutex_instances`.

By performing queries on both of the following tables, a monitoring application or a DBA can detect bottlenecks or deadlocks between threads that involve mutexes:

- `events_waits_current`, to see what mutex a thread is waiting for
- `mutex_instances`, to see which other thread currently owns a mutex

10.3.4 The rwlock_instances Table

The `rwlock_instances` table lists all the rwlock (read write lock) instances seen by the Performance Schema while the server executes. An `rwlock` is a synchronization mechanism used in the code to enforce that threads at a given time can have access to some common resource following certain rules. The resource is said to be "protected" by the `rwlock`. The access is either shared (many threads can have a read lock at the same time), exclusive (only one thread can have a write lock at a given time), or shared-exclusive (a thread can have a write lock while permitting inconsistent reads by other threads). Shared-exclusive access is otherwise known as an `sxlock` and optimizes concurrency and improves scalability for read-write workloads.

Depending on how many threads are requesting a lock, and the nature of the locks requested, access can be either granted in shared mode, exclusive mode, shared-exclusive mode or not granted at all, waiting for other threads to finish first.

The `rwlock_instances` table has these columns:

- `NAME`

  The instrument name associated with the lock.

- `OBJECT_INSTANCE_BEGIN`

  The address in memory of the instrumented lock.

- `WRITE_LOCKED_BY_THREAD_ID`

  When a thread currently has an `rwlock` locked in exclusive (write) mode, `WRITE_LOCKED_BY_THREAD_ID` is the `THREAD_ID` of the locking thread, otherwise it is `NULL`.

- `READ_LOCKED_BY_COUNT`

  When a thread currently has an `rwlock` locked in shared (read) mode, `READ_LOCKED_BY_COUNT` is incremented by 1. This is a counter only, so it cannot be used directly to find which thread holds a read lock, but it can be used to see whether there is a read contention on an `rwlock`, and see how many readers are currently active.

The `rwlock_instances` table has these indexes:

- Primary key on (`OBJECT_INSTANCE_BEGIN`)

- Index on (NAME)
- Index on (WRITE_LOCKED_BY_THREAD_ID)

TRUNCATE TABLE is not permitted for the rwlock_instances table.

By performing queries on both of the following tables, a monitoring application or a DBA may detect some bottlenecks or deadlocks between threads that involve locks:

- events_waits_current, to see what rwlock a thread is waiting for
- rwlock_instances, to see which other thread currently owns an rwlock

There is a limitation: The rwlock_instances can be used only to identify the thread holding a write lock, but not the threads holding a read lock.

10.3.5 The socket_instances Table

The socket_instances table provides a real-time snapshot of the active connections to the MySQL server. The table contains one row per TCP/IP or Unix socket file connection. Information available in this table provides a real-time snapshot of the active connections to the server. (Additional information is available in socket summary tables, including network activity such as socket operations and number of bytes transmitted and received; see Section 10.20.9, “Socket Summary Tables”).

```
mysql> SELECT * FROM performance_schema.socket_instances\G
*************************** 1. row ***************************
           EVENT_NAME: wait/io/socket/sql/server_unix_socket
OBJECT_INSTANCE_BEGIN: 4316619408
            THREAD_ID: 1
            SOCKET_ID: 16
                   IP:
                 PORT: 0
                STATE: ACTIVE
*************************** 2. row ***************************
           EVENT_NAME: wait/io/socket/sql/client_connection
OBJECT_INSTANCE_BEGIN: 4316644608
            THREAD_ID: 21
            SOCKET_ID: 39
                   IP: 127.0.0.1
                 PORT: 55233
                STATE: ACTIVE
*************************** 3. row ***************************
           EVENT_NAME: wait/io/socket/sql/server_tcpip_socket
OBJECT_INSTANCE_BEGIN: 4316699040
            THREAD_ID: 1
            SOCKET_ID: 14
                   IP: 0.0.0.0
                 PORT: 50603
                STATE: ACTIVE
```

Socket instruments have names of the form wait/io/socket/sql/*socket_type* and are used like this:

1. The server has a listening socket for each network protocol that it supports. The instruments associated with listening sockets for TCP/IP or Unix socket file connections have a *socket_type* value of server_tcpip_socket or server_unix_socket, respectively.
2. When a listening socket detects a connection, the server transfers the connection to a new socket managed by a separate thread. The instrument for the new connection thread has a *socket_type* value of client_connection.
3. When a connection terminates, the row in socket_instances corresponding to it is deleted.

The `socket_instances` table has these columns:

- `EVENT_NAME`

  The name of the `wait/io/socket/*` instrument that produced the event. This is a `NAME` value from the `setup_instruments` table. Instrument names may have multiple parts and form a hierarchy, as discussed in Chapter 7, *Performance Schema Instrument Naming Conventions*.

- `OBJECT_INSTANCE_BEGIN`

  This column uniquely identifies the socket. The value is the address of an object in memory.

- `THREAD_ID`

  The internal thread identifier assigned by the server. Each socket is managed by a single thread, so each socket can be mapped to a thread which can be mapped to a server process.

- `SOCKET_ID`

  The internal file handle assigned to the socket.

- `IP`

  The client IP address. The value may be either an IPv4 or IPv6 address, or blank to indicate a Unix socket file connection.

- `PORT`

  The TCP/IP port number, in the range from 0 to 65535.

- `STATE`

  The socket status, either `IDLE` or `ACTIVE`. Wait times for active sockets are tracked using the corresponding socket instrument. Wait times for idle sockets are tracked using the `idle` instrument.

  A socket is idle if it is waiting for a request from the client. When a socket becomes idle, the event row in `socket_instances` that is tracking the socket switches from a status of `ACTIVE` to `IDLE`. The `EVENT_NAME` value remains `wait/io/socket/*`, but timing for the instrument is suspended. Instead, an event is generated in the `events_waits_current` table with an `EVENT_NAME` value of `idle`.

  When the next request is received, the `idle` event terminates, the socket instance switches from `IDLE` to `ACTIVE`, and timing of the socket instrument resumes.

The `socket_instances` table has these indexes:

- Primary key on (`OBJECT_INSTANCE_BEGIN`)
- Index on (`THREAD_ID`)
- Index on (`SOCKET_ID`)
- Index on (`IP`, `PORT`)

`TRUNCATE TABLE` is not permitted for the `socket_instances` table.

The `IP:PORT` column combination value identifies the connection. This combination value is used in the `OBJECT_NAME` column of the `events_waits_xxx` tables, to identify the connection from which socket events come:

- For the Unix domain listener socket (`server_unix_socket`), the port is 0, and the IP is `''`.

- For client connections via the Unix domain listener (`client_connection`), the port is 0, and the IP is `''`.
- For the TCP/IP server listener socket (`server_tcpip_socket`), the port is always the master port (for example, 3306), and the IP is always `0.0.0.0`.
- For client connections via the TCP/IP listener (`client_connection`), the port is whatever the server assigns, but never 0. The IP is the IP of the originating host (`127.0.0.1` or `::1` for the local host)

10.4 Performance Schema Wait Event Tables

The Performance Schema instruments waits, which are events that take time. Within the event hierarchy, wait events nest within stage events, which nest within statement events, which nest within transaction events.

These tables store wait events:

- `events_waits_current`: The current wait event for each thread.
- `events_waits_history`: The most recent wait events that have ended per thread.
- `events_waits_history_long`: The most recent wait events that have ended globally (across all threads).

The following sections describe the wait event tables. There are also summary tables that aggregate information about wait events; see Section 10.20.1, “Wait Event Summary Tables”.

For more information about the relationship between the three wait event tables, see Performance Schema Tables for Current and Historical Events.

Configuring Wait Event Collection

To control whether to collect wait events, set the state of the relevant instruments and consumers:

- The `setup_instruments` table contains instruments with names that begin with `wait`. Use these instruments to enable or disable collection of individual wait event classes.
- The `setup_consumers` table contains consumer values with names corresponding to the current and historical wait event table names. Use these consumers to filter collection of wait events.

Some wait instruments are enabled by default; others are disabled. For example:

```
mysql> SELECT NAME, ENABLED, TIMED
       FROM performance_schema.setup_instruments
       WHERE NAME LIKE 'wait/io/file/innodb%';
+----------------------------------------------------+---------+-------+
| NAME                                               | ENABLED | TIMED |
+----------------------------------------------------+---------+-------+
| wait/io/file/innodb/innodb_tablespace_open_file | YES     | YES   |
| wait/io/file/innodb/innodb_data_file            | YES     | YES   |
| wait/io/file/innodb/innodb_log_file             | YES     | YES   |
| wait/io/file/innodb/innodb_temp_file            | YES     | YES   |
| wait/io/file/innodb/innodb_arch_file            | YES     | YES   |
| wait/io/file/innodb/innodb_clone_file           | YES     | YES   |
+----------------------------------------------------+---------+-------+
mysql> SELECT NAME, ENABLED, TIMED
       FROM performance_schema.setup_instruments
       WHERE NAME LIKE 'wait/io/socket/%';
+---------------------------------------+---------+-------+
| NAME                                  | ENABLED | TIMED |
+---------------------------------------+---------+-------+
| wait/io/socket/sql/server_tcpip_socket | NO      | NO    |
```

```
| wait/io/socket/sql/server_unix_socket  | NO      | NO    |
| wait/io/socket/sql/client_connection   | NO      | NO    |
+----------------------------------------+---------+-------+
```

The wait consumers are disabled by default:

```
mysql> SELECT *
       FROM performance_schema.setup_consumers
       WHERE NAME LIKE 'events_waits%';
+--------------------------+---------+
| NAME                     | ENABLED |
+--------------------------+---------+
| events_waits_current     | NO      |
| events_waits_history     | NO      |
| events_waits_history_long | NO      |
+--------------------------+---------+
```

To control wait event collection at server startup, use lines like these in your `my.cnf` file:

- Enable:

```
[mysqld]
performance-schema-instrument='wait/%=ON'
performance-schema-consumer-events-waits-current=ON
performance-schema-consumer-events-waits-history=ON
performance-schema-consumer-events-waits-history-long=ON
```

- Disable:

```
[mysqld]
performance-schema-instrument='wait/%=OFF'
performance-schema-consumer-events-waits-current=OFF
performance-schema-consumer-events-waits-history=OFF
performance-schema-consumer-events-waits-history-long=OFF
```

To control wait event collection at runtime, update the `setup_instruments` and `setup_consumers` tables:

- Enable:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'YES', TIMED = 'YES'
WHERE NAME LIKE 'wait/%';
UPDATE performance_schema.setup_consumers
SET ENABLED = 'YES'
WHERE NAME LIKE 'events_waits%';
```

- Disable:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'NO', TIMED = 'NO'
WHERE NAME LIKE 'wait/%';
UPDATE performance_schema.setup_consumers
SET ENABLED = 'NO'
WHERE NAME LIKE 'events_waits%';
```

To collect only specific wait events, enable only the corresponding wait instruments. To collect wait events only for specific wait event tables, enable the wait instruments but only the wait consumers corresponding to the desired tables.

For additional information about configuring event collection, see Chapter 4, *Performance Schema Startup Configuration*, and Chapter 5, *Performance Schema Runtime Configuration*.

10.4.1 The events_waits_current Table

The events_waits_current table contains current wait events. The table stores one row per thread showing the current status of the thread's most recent monitored wait event, so there is no system variable for configuring the table size.

Of the tables that contain wait event rows, events_waits_current is the most fundamental. Other tables that contain wait event rows are logically derived from the current events. For example, the events_waits_history and events_waits_history_long tables are collections of the most recent wait events that have ended, up to a maximum number of rows per thread and globally across all threads, respectively.

For more information about the relationship between the three wait event tables, see Performance Schema Tables for Current and Historical Events.

For information about configuring whether to collect wait events, see Section 10.4, “Performance Schema Wait Event Tables”.

The events_waits_current table has these columns:

- THREAD_ID, EVENT_ID

  The thread associated with the event and the thread current event number when the event starts. The THREAD_ID and EVENT_ID values taken together uniquely identify the row. No two rows have the same pair of values.

- END_EVENT_ID

  This column is set to NULL when the event starts and updated to the thread current event number when the event ends.

- EVENT_NAME

  The name of the instrument that produced the event. This is a NAME value from the setup_instruments table. Instrument names may have multiple parts and form a hierarchy, as discussed in Chapter 7, *Performance Schema Instrument Naming Conventions*.

- SOURCE

  The name of the source file containing the instrumented code that produced the event and the line number in the file at which the instrumentation occurs. This enables you to check the source to determine exactly what code is involved. For example, if a mutex or lock is being blocked, you can check the context in which this occurs.

- TIMER_START, TIMER_END, TIMER_WAIT

  Timing information for the event. The unit for these values is picoseconds (trillionths of a second). The TIMER_START and TIMER_END values indicate when event timing started and ended. TIMER_WAIT is the event elapsed time (duration).

  If an event has not finished, TIMER_END is the current timer value and TIMER_WAIT is the time elapsed so far (TIMER_END − TIMER_START).

  If an event is produced from an instrument that has TIMED = NO, timing information is not collected, and TIMER_START, TIMER_END, and TIMER_WAIT are all NULL.

  For discussion of picoseconds as the unit for event times and factors that affect time values, see Section 5.1, “Performance Schema Event Timing”.

- SPINS

For a mutex, the number of spin rounds. If the value is `NULL`, the code does not use spin rounds or spinning is not instrumented.

- `OBJECT_SCHEMA`, `OBJECT_NAME`, `OBJECT_TYPE`, `OBJECT_INSTANCE_BEGIN`

  These columns identify the object "being acted on." What that means depends on the object type.

  For a synchronization object (`cond`, `mutex`, `rwlock`):

  - `OBJECT_SCHEMA`, `OBJECT_NAME`, and `OBJECT_TYPE` are `NULL`.
  - `OBJECT_INSTANCE_BEGIN` is the address of the synchronization object in memory.

  For a file I/O object:

  - `OBJECT_SCHEMA` is `NULL`.
  - `OBJECT_NAME` is the file name.
  - `OBJECT_TYPE` is `FILE`.
  - `OBJECT_INSTANCE_BEGIN` is an address in memory.

  For a socket object:

  - `OBJECT_NAME` is the `IP:PORT` value for the socket.
  - `OBJECT_INSTANCE_BEGIN` is an address in memory.

  For a table I/O object:

  - `OBJECT_SCHEMA` is the name of the schema that contains the table.
  - `OBJECT_NAME` is the table name.
  - `OBJECT_TYPE` is `TABLE` for a persistent base table or `TEMPORARY TABLE` for a temporary table.
  - `OBJECT_INSTANCE_BEGIN` is an address in memory.

  An `OBJECT_INSTANCE_BEGIN` value itself has no meaning, except that different values indicate different objects. `OBJECT_INSTANCE_BEGIN` can be used for debugging. For example, it can be used with `GROUP BY OBJECT_INSTANCE_BEGIN` to see whether the load on 1,000 mutexes (that protect, say, 1,000 pages or blocks of data) is spread evenly or just hitting a few bottlenecks. This can help you correlate with other sources of information if you see the same object address in a log file or another debugging or performance tool.

- `INDEX_NAME`

  The name of the index used. `PRIMARY` indicates the table primary index. `NULL` means that no index was used.

- `NESTING_EVENT_ID`

  The `EVENT_ID` value of the event within which this event is nested.

- `NESTING_EVENT_TYPE`

  The nesting event type. The value is `TRANSACTION`, `STATEMENT`, `STAGE`, or `WAIT`.

- `OPERATION`

  The type of operation performed, such as `lock`, `read`, or `write`.

- `NUMBER_OF_BYTES`

  The number of bytes read or written by the operation. For table I/O waits (events for the `wait/io/table/sql/handler` instrument), `NUMBER_OF_BYTES` indicates the number of rows. If the value is greater than 1, the event is for a batch I/O operation. The following discussion describes the difference between exclusively single-row reporting and reporting that reflects batch I/O.

  MySQL executes joins using a nested-loop implementation. The job of the Performance Schema instrumentation is to provide row count and accumulated execution time per table in the join. Assume a join query of the following form that is executed using a table join order of `t1`, `t2`, `t3`:

  ```
  SELECT ... FROM t1 JOIN t2 ON ... JOIN t3 ON ...
  ```

  Table “fanout” is the increase or decrease in number of rows from adding a table during join processing. If the fanout for table `t3` is greater than 1, the majority of row-fetch operations are for that table. Suppose that the join accesses 10 rows from `t1`, 20 rows from `t2` per row from `t1`, and 30 rows from `t3` per row of table `t2`. With single-row reporting, the total number of instrumented operations is:

  ```
  10 + (10 * 20) + (10 * 20 * 30) = 6210
  ```

  A significant reduction in the number of instrumented operations is achievable by aggregating them per scan (that is, per unique combination of rows from `t1` and `t2`). With batch I/O reporting, the Performance Schema produces an event for each scan of the innermost table `t3` rather than for each row, and the number of instrumented row operations reduces to:

  ```
  10 + (10 * 20) + (10 * 20) = 410
  ```

  That is a reduction of 93%, illustrating how the batch-reporting strategy significantly reduces Performance Schema overhead for table I/O by reducing the number of reporting calls. The tradeoff is lesser accuracy for event timing. Rather than time for an individual row operation as in per-row reporting, timing for batch I/O includes time spent for operations such as join buffering, aggregation, and returning rows to the client.

  For batch I/O reporting to occur, these conditions must be true:

  - Query execution accesses the innermost table of a query block (for a single-table query, that table counts as innermost)
  - Query execution does not request a single row from the table (so, for example, `eq_ref` access prevents use of batch reporting)
  - Query execution does not evaluate a subquery containing table access for the table

- `FLAGS`

  Reserved for future use.

The `events_waits_current` table has these indexes:

- Primary key on (`THREAD_ID`, `EVENT_ID`)

`TRUNCATE TABLE` is permitted for the `events_waits_current` table. It removes the rows.

10.4.2 The events_waits_history Table

The `events_waits_history` table contains the *`N`* most recent wait events that have ended per thread. Wait events are not added to the table until they have ended. When the table contains the maximum number of rows for a given thread, the oldest thread row is discarded when a new row for that thread is added. When a thread ends, all its rows are discarded.

The Performance Schema autosizes the value of *`N`* during server startup. To set the number of rows per thread explicitly, set the `performance_schema_events_waits_history_size` system variable at server startup.

The `events_waits_history` table has the same columns and indexing as `events_waits_current`. See Section 10.4.1, “The events_waits_current Table”.

`TRUNCATE TABLE` is permitted for the `events_waits_history` table. It removes the rows.

For more information about the relationship between the three wait event tables, see Performance Schema Tables for Current and Historical Events.

For information about configuring whether to collect wait events, see Section 10.4, “Performance Schema Wait Event Tables”.

10.4.3 The events_waits_history_long Table

The `events_waits_history_long` table contains *`N`* the most recent wait events that have ended globally, across all threads. Wait events are not added to the table until they have ended. When the table becomes full, the oldest row is discarded when a new row is added, regardless of which thread generated either row.

The Performance Schema autosizes the value of *`N`* during server startup. To set the table size explicitly, set the `performance_schema_events_waits_history_long_size` system variable at server startup.

The `events_waits_history_long` table has the same columns as `events_waits_current`. See Section 10.4.1, “The events_waits_current Table”. Unlike `events_waits_current`, `events_waits_history_long` has no indexing.

`TRUNCATE TABLE` is permitted for the `events_waits_history_long` table. It removes the rows.

For more information about the relationship between the three wait event tables, see Performance Schema Tables for Current and Historical Events.

For information about configuring whether to collect wait events, see Section 10.4, “Performance Schema Wait Event Tables”.

10.5 Performance Schema Stage Event Tables

The Performance Schema instruments stages, which are steps during the statement-execution process, such as parsing a statement, opening a table, or performing a `filesort` operation. Stages correspond to the thread states displayed by `SHOW PROCESSLIST` or that are visible in the Information Schema `PROCESSLIST` table. Stages begin and end when state values change.

Within the event hierarchy, wait events nest within stage events, which nest within statement events, which nest within transaction events.

These tables store stage events:

- `events_stages_current`: The current stage event for each thread.
- `events_stages_history`: The most recent stage events that have ended per thread.

- `events_stages_history_long`: The most recent stage events that have ended globally (across all threads).

The following sections describe the stage event tables. There are also summary tables that aggregate information about stage events; see Section 10.20.2, “Stage Summary Tables”.

For more information about the relationship between the three stage event tables, see Performance Schema Tables for Current and Historical Events.

- Configuring Stage Event Collection
- Stage Event Progress Information

Configuring Stage Event Collection

To control whether to collect stage events, set the state of the relevant instruments and consumers:

- The `setup_instruments` table contains instruments with names that begin with `stage`. Use these instruments to enable or disable collection of individual stage event classes.
- The `setup_consumers` table contains consumer values with names corresponding to the current and historical stage event table names. Use these consumers to filter collection of stage events.

Other than those instruments that provide statement progress information, the stage instruments are disabled by default. For example:

```
mysql> SELECT NAME, ENABLED, TIMED
       FROM performance_schema.setup_instruments
       WHERE NAME RLIKE 'stage/sql/[a-c]';
+-----------------------------------------------------+---------+-------+
| NAME                                                | ENABLED | TIMED |
+-----------------------------------------------------+---------+-------+
| stage/sql/After create                              | NO      | NO    |
| stage/sql/allocating local table                    | NO      | NO    |
| stage/sql/altering table                            | NO      | NO    |
| stage/sql/committing alter table to storage engine | NO      | NO    |
| stage/sql/Changing master                           | NO      | NO    |
| stage/sql/Checking master version                   | NO      | NO    |
| stage/sql/checking permissions                      | NO      | NO    |
| stage/sql/cleaning up                               | NO      | NO    |
| stage/sql/closing tables                            | NO      | NO    |
| stage/sql/Connecting to master                      | NO      | NO    |
| stage/sql/converting HEAP to MyISAM                 | NO      | NO    |
| stage/sql/Copying to group table                    | NO      | NO    |
| stage/sql/Copying to tmp table                      | NO      | NO    |
| stage/sql/copy to tmp table                         | NO      | NO    |
| stage/sql/Creating sort index                       | NO      | NO    |
| stage/sql/creating table                            | NO      | NO    |
| stage/sql/Creating tmp table                        | NO      | NO    |
+-----------------------------------------------------+---------+-------+
```

Stage event instruments that provide statement progress information are enabled and timed by default:

```
mysql> SELECT NAME, ENABLED, TIMED
       FROM performance_schema.setup_instruments
       WHERE ENABLED='YES' AND NAME LIKE "stage/%";
+-----------------------------------------------------+---------+-------+
| NAME                                                | ENABLED | TIMED |
+-----------------------------------------------------+---------+-------+
| stage/sql/copy to tmp table                         | YES     | YES   |
| stage/sql/Applying batch of row changes (write)     | YES     | YES   |
| stage/sql/Applying batch of row changes (update)    | YES     | YES   |
```

```
| stage/sql/Applying batch of row changes (delete)     | YES     | YES   |
| stage/innodb/alter table (end)                       | YES     | YES   |
| stage/innodb/alter table (flush)                     | YES     | YES   |
| stage/innodb/alter table (insert)                    | YES     | YES   |
| stage/innodb/alter table (log apply index)           | YES     | YES   |
| stage/innodb/alter table (log apply table)           | YES     | YES   |
| stage/innodb/alter table (merge sort)                | YES     | YES   |
| stage/innodb/alter table (read PK and internal sort) | YES     | YES   |
| stage/innodb/buffer pool load                        | YES     | YES   |
| stage/innodb/clone (file copy)                       | YES     | YES   |
| stage/innodb/clone (redo copy)                       | YES     | YES   |
| stage/innodb/clone (page copy)                       | YES     | YES   |
+------------------------------------------------------+---------+-------+
```

The stage consumers are disabled by default:

```
mysql> SELECT *
       FROM performance_schema.setup_consumers
       WHERE NAME LIKE 'events_stages%';
+----------------------------+---------+
| NAME                       | ENABLED |
+----------------------------+---------+
| events_stages_current      | NO      |
| events_stages_history      | NO      |
| events_stages_history_long | NO      |
+----------------------------+---------+
```

To control stage event collection at server startup, use lines like these in your `my.cnf` file:

- Enable:

  ```
  [mysqld]
  performance-schema-instrument='stage/%=ON'
  performance-schema-consumer-events-stages-current=ON
  performance-schema-consumer-events-stages-history=ON
  performance-schema-consumer-events-stages-history-long=ON
  ```

- Disable:

  ```
  [mysqld]
  performance-schema-instrument='stage/%=OFF'
  performance-schema-consumer-events-stages-current=OFF
  performance-schema-consumer-events-stages-history=OFF
  performance-schema-consumer-events-stages-history-long=OFF
  ```

To control stage event collection at runtime, update the `setup_instruments` and `setup_consumers` tables:

- Enable:

  ```
  UPDATE performance_schema.setup_instruments
  SET ENABLED = 'YES', TIMED = 'YES'
  WHERE NAME LIKE 'stage/%';
  UPDATE performance_schema.setup_consumers
  SET ENABLED = 'YES'
  WHERE NAME LIKE 'events_stages%';
  ```

- Disable:

  ```
  UPDATE performance_schema.setup_instruments
  SET ENABLED = 'NO', TIMED = 'NO'
  WHERE NAME LIKE 'stage/%';
  UPDATE performance_schema.setup_consumers
  SET ENABLED = 'NO'
  WHERE NAME LIKE 'events_stages%';
  ```

To collect only specific stage events, enable only the corresponding stage instruments. To collect stage events only for specific stage event tables, enable the stage instruments but only the stage consumers corresponding to the desired tables.

For additional information about configuring event collection, see Chapter 4, *Performance Schema Startup Configuration*, and Chapter 5, *Performance Schema Runtime Configuration*.

Stage Event Progress Information

The Performance Schema stage event tables contain two columns that, taken together, provide a stage progress indicator for each row:

- `WORK_COMPLETED`: The number of work units completed for the stage
- `WORK_ESTIMATED`: The number of work units expected for the stage

Each column is `NULL` if no progress information is provided for an instrument. Interpretation of the information, if it is available, depends entirely on the instrument implementation. The Performance Schema tables provide a container to store progress data, but make no assumptions about the semantics of the metric itself:

- A “work unit” is an integer metric that increases over time during execution, such as the number of bytes, rows, files, or tables processed. The definition of “work unit” for a particular instrument is left to the instrumentation code providing the data.
- The `WORK_COMPLETED` value can increase one or many units at a time, depending on the instrumented code.
- The `WORK_ESTIMATED` value can change during the stage, depending on the instrumented code.

Instrumentation for a stage event progress indicator can implement any of the following behaviors:

- No progress instrumentation

  This is the most typical case, where no progress data is provided. The `WORK_COMPLETED` and `WORK_ESTIMATED` columns are both `NULL`.

- Unbounded progress instrumentation

  Only the `WORK_COMPLETED` column is meaningful. No data is provided for the `WORK_ESTIMATED` column, which displays 0.

  By querying the `events_stages_current` table for the monitored session, a monitoring application can report how much work has been performed so far, but cannot report whether the stage is near completion. Currently, no stages are instrumented like this.

- Bounded progress instrumentation

  The `WORK_COMPLETED` and `WORK_ESTIMATED` columns are both meaningful.

  This type of progress indicator is appropriate for an operation with a defined completion criterion, such as the table-copy instrument described later. By querying the `events_stages_current` table for the monitored session, a monitoring application can report how much work has been performed so far, and can report the overall completion percentage for the stage, by computing the `WORK_COMPLETED` / `WORK_ESTIMATED` ratio.

The `stage/sql/copy to tmp table` instrument illustrates how progress indicators work. During execution of an `ALTER TABLE` statement, the `stage/sql/copy to tmp table` stage is used, and this stage can execute potentially for a long time, depending on the size of the data to copy.

The table-copy task has a defined termination (all rows copied), and the `stage/sql/copy to tmp table` stage is instrumented to provided bounded progress information: The work unit used is number of rows copied, `WORK_COMPLETED` and `WORK_ESTIMATED` are both meaningful, and their ratio indicates task percentage complete.

To enable the instrument and the relevant consumers, execute these statements:

```
UPDATE performance_schema.setup_instruments
SET ENABLED='YES'
WHERE NAME='stage/sql/copy to tmp table';
UPDATE performance_schema.setup_consumers
SET ENABLED='YES'
WHERE NAME LIKE 'events_stages_%';
```

To see the progress of an ongoing `ALTER TABLE` statement, select from the `events_stages_current` table.

10.5.1 The events_stages_current Table

The `events_stages_current` table contains current stage events. The table stores one row per thread showing the current status of the thread's most recent monitored stage event, so there is no system variable for configuring the table size.

Of the tables that contain stage event rows, `events_stages_current` is the most fundamental. Other tables that contain stage event rows are logically derived from the current events. For example, the `events_stages_history` and `events_stages_history_long` tables are collections of the most recent stage events that have ended, up to a maximum number of rows per thread and globally across all threads, respectively.

For more information about the relationship between the three stage event tables, see Performance Schema Tables for Current and Historical Events.

For information about configuring whether to collect stage events, see Section 10.5, “Performance Schema Stage Event Tables”.

The `events_stages_current` table has these columns:

- `THREAD_ID`, `EVENT_ID`

  The thread associated with the event and the thread current event number when the event starts. The `THREAD_ID` and `EVENT_ID` values taken together uniquely identify the row. No two rows have the same pair of values.

- `END_EVENT_ID`

  This column is set to `NULL` when the event starts and updated to the thread current event number when the event ends.

- `EVENT_NAME`

  The name of the instrument that produced the event. This is a `NAME` value from the `setup_instruments` table. Instrument names may have multiple parts and form a hierarchy, as discussed in Chapter 7, *Performance Schema Instrument Naming Conventions*.

- `SOURCE`

  The name of the source file containing the instrumented code that produced the event and the line number in the file at which the instrumentation occurs. This enables you to check the source to determine exactly what code is involved.

- `TIMER_START`, `TIMER_END`, `TIMER_WAIT`

  Timing information for the event. The unit for these values is picoseconds (trillionths of a second). The `TIMER_START` and `TIMER_END` values indicate when event timing started and ended. `TIMER_WAIT` is the event elapsed time (duration).

  If an event has not finished, `TIMER_END` is the current timer value and `TIMER_WAIT` is the time elapsed so far (`TIMER_END` − `TIMER_START`).

  If an event is produced from an instrument that has `TIMED = NO`, timing information is not collected, and `TIMER_START`, `TIMER_END`, and `TIMER_WAIT` are all `NULL`.

  For discussion of picoseconds as the unit for event times and factors that affect time values, see Section 5.1, “Performance Schema Event Timing”.

- `WORK_COMPLETED`, `WORK_ESTIMATED`

  These columns provide stage progress information, for instruments that have been implemented to produce such information. `WORK_COMPLETED` indicates how many work units have been completed for the stage, and `WORK_ESTIMATED` indicates how many work units are expected for the stage. For more information, see Stage Event Progress Information.

- `NESTING_EVENT_ID`

  The `EVENT_ID` value of the event within which this event is nested. The nesting event for a stage event is usually a statement event.

- `NESTING_EVENT_TYPE`

  The nesting event type. The value is `TRANSACTION`, `STATEMENT`, `STAGE`, or `WAIT`.

The `events_stages_current` table has these indexes:

- Primary key on (`THREAD_ID`, `EVENT_ID`)

`TRUNCATE TABLE` is permitted for the `events_stages_current` table. It removes the rows.

10.5.2 The events_stages_history Table

The `events_stages_history` table contains the *`N`* most recent stage events that have ended per thread. Stage events are not added to the table until they have ended. When the table contains the maximum number of rows for a given thread, the oldest thread row is discarded when a new row for that thread is added. When a thread ends, all its rows are discarded.

The Performance Schema autosizes the value of *`N`* during server startup. To set the number of rows per thread explicitly, set the `performance_schema_events_stages_history_size` system variable at server startup.

The `events_stages_history` table has the same columns and indexing as `events_stages_current`. See Section 10.5.1, “The events_stages_current Table”.

`TRUNCATE TABLE` is permitted for the `events_stages_history` table. It removes the rows.

For more information about the relationship between the three stage event tables, see Performance Schema Tables for Current and Historical Events.

For information about configuring whether to collect stage events, see Section 10.5, “Performance Schema Stage Event Tables”.

10.5.3 The events_stages_history_long Table

The `events_stages_history_long` table contains the *`N`* most recent stage events that have ended globally, across all threads. Stage events are not added to the table until they have ended. When the table becomes full, the oldest row is discarded when a new row is added, regardless of which thread generated either row.

The Performance Schema autosizes the value of *`N`* during server startup. To set the table size explicitly, set the `performance_schema_events_stages_history_long_size` system variable at server startup.

The `events_stages_history_long` table has the same columns as `events_stages_current`. See Section 10.5.1, “The events_stages_current Table”. Unlike `events_stages_current`, `events_stages_history_long` has no indexing.

`TRUNCATE TABLE` is permitted for the `events_stages_history_long` table. It removes the rows.

For more information about the relationship between the three stage event tables, see Performance Schema Tables for Current and Historical Events.

For information about configuring whether to collect stage events, see Section 10.5, “Performance Schema Stage Event Tables”.

10.6 Performance Schema Statement Event Tables

The Performance Schema instruments statement execution. Statement events occur at a high level of the event hierarchy. Within the event hierarchy, wait events nest within stage events, which nest within statement events, which nest within transaction events.

These tables store statement events:

- `events_statements_current`: The current statement event for each thread.
- `events_statements_history`: The most recent statement events that have ended per thread.
- `events_statements_history_long`: The most recent statement events that have ended globally (across all threads).
- `prepared_statements_instances`: Prepared statement instances and statistics

The following sections describe the statement event tables. There are also summary tables that aggregate information about statement events; see Section 10.20.3, “Statement Summary Tables”.

For more information about the relationship between the three `events_statements_xxx` event tables, see Performance Schema Tables for Current and Historical Events.

- Configuring Statement Event Collection
- Statement Monitoring

Configuring Statement Event Collection

To control whether to collect statement events, set the state of the relevant instruments and consumers:

- The `setup_instruments` table contains instruments with names that begin with `statement`. Use these instruments to enable or disable collection of individual statement event classes.
- The `setup_consumers` table contains consumer values with names corresponding to the current and historical statement event table names, and the statement digest consumer. Use these consumers to filter collection of statement events and statement digesting.

The statement instruments are enabled by default, and the `events_statements_current`, `events_statements_history`, and `statements_digest` statement consumers are enabled by default:

```
mysql> SELECT NAME, ENABLED, TIMED
       FROM performance_schema.setup_instruments
       WHERE NAME LIKE 'statement/%';
+----------------------------------------+---------+-------+
| NAME                                   | ENABLED | TIMED |
+----------------------------------------+---------+-------+
| statement/sql/select                   | YES     | YES   |
| statement/sql/create_table             | YES     | YES   |
| statement/sql/create_index             | YES     | YES   |
...
| statement/sp/stmt                      | YES     | YES   |
| statement/sp/set                       | YES     | YES   |
| statement/sp/set_trigger_field         | YES     | YES   |
| statement/scheduler/event              | YES     | YES   |
| statement/com/Sleep                    | YES     | YES   |
| statement/com/Quit                     | YES     | YES   |
| statement/com/Init DB                  | YES     | YES   |
...
| statement/abstract/Query               | YES     | YES   |
| statement/abstract/new_packet          | YES     | YES   |
| statement/abstract/relay_log           | YES     | YES   |
+----------------------------------------+---------+-------+
```

```
mysql> SELECT *
       FROM performance_schema.setup_consumers
       WHERE NAME LIKE '%statements%';
+--------------------------------+---------+
| NAME                           | ENABLED |
+--------------------------------+---------+
| events_statements_current      | YES     |
| events_statements_history      | YES     |
| events_statements_history_long | NO      |
| statements_digest              | YES     |
+--------------------------------+---------+
```

To control statement event collection at server startup, use lines like these in your `my.cnf` file:

- Enable:

  ```
  [mysqld]
  performance-schema-instrument='statement/%=ON'
  performance-schema-consumer-events-statements-current=ON
  performance-schema-consumer-events-statements-history=ON
  performance-schema-consumer-events-statements-history-long=ON
  performance-schema-consumer-statements-digest=ON
  ```

- Disable:

  ```
  [mysqld]
  performance-schema-instrument='statement/%=OFF'
  performance-schema-consumer-events-statements-current=OFF
  performance-schema-consumer-events-statements-history=OFF
  performance-schema-consumer-events-statements-history-long=OFF
  performance-schema-consumer-statements-digest=OFF
  ```

To control statement event collection at runtime, update the `setup_instruments` and `setup_consumers` tables:

- Enable:

  ```
  UPDATE performance_schema.setup_instruments
  ```

```
SET ENABLED = 'YES', TIMED = 'YES'
WHERE NAME LIKE 'statement/%';
UPDATE performance_schema.setup_consumers
SET ENABLED = 'YES'
WHERE NAME LIKE '%statements%';
```

- Disable:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'NO', TIMED = 'NO'
WHERE NAME LIKE 'statement/%';
UPDATE performance_schema.setup_consumers
SET ENABLED = 'NO'
WHERE NAME LIKE '%statements%';
```

To collect only specific statement events, enable only the corresponding statement instruments. To collect statement events only for specific statement event tables, enable the statement instruments but only the statement consumers corresponding to the desired tables.

For additional information about configuring event collection, see Chapter 4, *Performance Schema Startup Configuration*, and Chapter 5, *Performance Schema Runtime Configuration*.

Statement Monitoring

Statement monitoring begins from the moment the server sees that activity is requested on a thread, to the moment when all activity has ceased. Typically, this means from the time the server gets the first packet from the client to the time the server has finished sending the response. Statements within stored programs are monitored like other statements.

When the Performance Schema instruments a request (server command or SQL statement), it uses instrument names that proceed in stages from more general (or “abstract”) to more specific until it arrives at a final instrument name.

Final instrument names correspond to server commands and SQL statements:

- Server commands correspond to the `COM_xxx codes` defined in the `mysql_com.h` header file and processed in `sql/sql_parse.cc`. Examples are `COM_PING` and `COM_QUIT`. Instruments for commands have names that begin with `statement/com`, such as `statement/com/Ping` and `statement/com/Quit`.
- SQL statements are expressed as text, such as `DELETE FROM t1` or `SELECT * FROM t2`. Instruments for SQL statements have names that begin with `statement/sql`, such as `statement/sql/delete` and `statement/sql/select`.

Some final instrument names are specific to error handling:

- `statement/com/Error` accounts for messages received by the server that are out of band. It can be used to detect commands sent by clients that the server does not understand. This may be helpful for purposes such as identifying clients that are misconfigured or using a version of MySQL more recent than that of the server, or clients that are attempting to attack the server.
- `statement/sql/error` accounts for SQL statements that fail to parse. It can be used to detect malformed queries sent by clients. A query that fails to parse differs from a query that parses but fails due to an error during execution. For example, `SELECT * FROM` is malformed, and the `statement/sql/error` instrument is used. By contrast, `SELECT *` parses but fails with a `No tables used` error. In this case, `statement/sql/select` is used and the statement event contains information to indicate the nature of the error.

A request can be obtained from any of these sources:

- As a command or statement request from a client, which sends the request as packets
- As a statement string read from the relay log on a replica
- As an event from the Event Scheduler

The details for a request are not initially known and the Performance Schema proceeds from abstract to specific instrument names in a sequence that depends on the source of the request.

For a request received from a client:

1. When the server detects a new packet at the socket level, a new statement is started with an abstract instrument name of `statement/abstract/new_packet`.
2. When the server reads the packet number, it knows more about the type of request received, and the Performance Schema refines the instrument name. For example, if the request is a `COM_PING` packet, the instrument name becomes `statement/com/Ping` and that is the final name. If the request is a `COM_QUERY` packet, it is known to correspond to an SQL statement but not the particular type of statement. In this case, the instrument changes from one abstract name to a more specific but still abstract name, `statement/abstract/Query`, and the request requires further classification.
3. If the request is a statement, the statement text is read and given to the parser. After parsing, the exact statement type is known. If the request is, for example, an `INSERT` statement, the Performance Schema refines the instrument name from `statement/abstract/Query` to `statement/sql/insert`, which is the final name.

For a request read as a statement from the relay log on a replica:

1. Statements in the relay log are stored as text and are read as such. There is no network protocol, so the `statement/abstract/new_packet` instrument is not used. Instead, the initial instrument is `statement/abstract/relay_log`.
2. When the statement is parsed, the exact statement type is known. If the request is, for example, an `INSERT` statement, the Performance Schema refines the instrument name from `statement/abstract/Query` to `statement/sql/insert`, which is the final name.

The preceding description applies only for statement-based replication. For row-based replication, table I/O done on the replica as it processes row changes can be instrumented, but row events in the relay log do not appear as discrete statements.

For a request received from the Event Scheduler:

The event execution is instrumented using the name `statement/scheduler/event`. This is the final name.

Statements executed within the event body are instrumented using `statement/sql/*` names, without use of any preceding abstract instrument. An event is a stored program, and stored programs are precompiled in memory before execution. Consequently, there is no parsing at runtime and the type of each statement is known by the time it executes.

Statements executed within the event body are child statements. For example, if an event executes an `INSERT` statement, execution of the event itself is the parent, instrumented using `statement/scheduler/event`, and the `INSERT` is the child, instrumented using `statement/sql/insert`. The parent/child relationship holds *between* separate instrumented operations. This differs from the sequence of refinement that occurs *within* a single instrumented operation, from abstract to final instrument names.

For statistics to be collected for statements, it is not sufficient to enable only the final `statement/sql/*` instruments used for individual statement types. The abstract `statement/abstract/*` instruments must

be enabled as well. This should not normally be an issue because all statement instruments are enabled by default. However, an application that enables or disables statement instruments selectively must take into account that disabling abstract instruments also disables statistics collection for the individual statement instruments. For example, to collect statistics for `INSERT` statements, `statement/sql/insert` must be enabled, but also `statement/abstract/new_packet` and `statement/abstract/Query`. Similarly, for replicated statements to be instrumented, `statement/abstract/relay_log` must be enabled.

No statistics are aggregated for abstract instruments such as `statement/abstract/Query` because no statement is ever classified with an abstract instrument as the final statement name.

10.6.1 The events_statements_current Table

The `events_statements_current` table contains current statement events. The table stores one row per thread showing the current status of the thread's most recent monitored statement event, so there is no system variable for configuring the table size.

Of the tables that contain statement event rows, `events_statements_current` is the most fundamental. Other tables that contain statement event rows are logically derived from the current events. For example, the `events_statements_history` and `events_statements_history_long` tables are collections of the most recent statement events that have ended, up to a maximum number of rows per thread and globally across all threads, respectively.

For more information about the relationship between the three `events_statements_xxx` event tables, see Performance Schema Tables for Current and Historical Events.

For information about configuring whether to collect statement events, see Section 10.6, “Performance Schema Statement Event Tables”.

The `events_statements_current` table has these columns:

- `THREAD_ID`, `EVENT_ID`

  The thread associated with the event and the thread current event number when the event starts. The `THREAD_ID` and `EVENT_ID` values taken together uniquely identify the row. No two rows have the same pair of values.

- `END_EVENT_ID`

  This column is set to `NULL` when the event starts and updated to the thread current event number when the event ends.

- `EVENT_NAME`

  The name of the instrument from which the event was collected. This is a `NAME` value from the `setup_instruments` table. Instrument names may have multiple parts and form a hierarchy, as discussed in Chapter 7, *Performance Schema Instrument Naming Conventions*.

  For SQL statements, the `EVENT_NAME` value initially is `statement/com/Query` until the statement is parsed, then changes to a more appropriate value, as described in Section 10.6, “Performance Schema Statement Event Tables”.

- `SOURCE`

  The name of the source file containing the instrumented code that produced the event and the line number in the file at which the instrumentation occurs. This enables you to check the source to determine exactly what code is involved.

- `TIMER_START`, `TIMER_END`, `TIMER_WAIT`

  Timing information for the event. The unit for these values is picoseconds (trillionths of a second). The `TIMER_START` and `TIMER_END` values indicate when event timing started and ended. `TIMER_WAIT` is the event elapsed time (duration).

  If an event has not finished, `TIMER_END` is the current timer value and `TIMER_WAIT` is the time elapsed so far (`TIMER_END` − `TIMER_START`).

  If an event is produced from an instrument that has `TIMED = NO`, timing information is not collected, and `TIMER_START`, `TIMER_END`, and `TIMER_WAIT` are all `NULL`.

  For discussion of picoseconds as the unit for event times and factors that affect time values, see Section 5.1, “Performance Schema Event Timing”.

- `LOCK_TIME`

  The time spent waiting for table locks. This value is computed in microseconds but normalized to picoseconds for easier comparison with other Performance Schema timers.

- `SQL_TEXT`

  The text of the SQL statement. For a command not associated with an SQL statement, the value is `NULL`.

  The maximum space available for statement display is 1024 bytes by default. To change this value, set the `performance_schema_max_sql_text_length` system variable at server startup. (Changing this value affects columns in other Performance Schema tables as well. See Performance Schema Statement Digests and Sampling.)

- `DIGEST`

  The statement digest SHA-256 value as a string of 64 hexadecimal characters, or `NULL` if the `statements_digest` consumer is `no`. For more information about statement digesting, see Performance Schema Statement Digests and Sampling.

- `DIGEST_TEXT`

  The normalized statement digest text, or `NULL` if the `statements_digest` consumer is `no`. For more information about statement digesting, see Performance Schema Statement Digests and Sampling.

  The `performance_schema_max_digest_length` system variable determines the maximum number of bytes available per session for digest value storage. However, the display length of statement digests may be longer than the available buffer size due to encoding of statement elements such as keywords and literal values in digest buffer. Consequently, values selected from the `DIGEST_TEXT` column of statement event tables may appear to exceed the `performance_schema_max_digest_length` value.

- `CURRENT_SCHEMA`

  The default database for the statement, `NULL` if there is none.

- `OBJECT_SCHEMA`, `OBJECT_NAME`, `OBJECT_TYPE`

  For nested statements (stored programs), these columns contain information about the parent statement. Otherwise they are `NULL`.

- `OBJECT_INSTANCE_BEGIN`

This column identifies the statement. The value is the address of an object in memory.

- `MYSQL_ERRNO`

  The statement error number, from the statement diagnostics area.

- `RETURNED_SQLSTATE`

  The statement SQLSTATE value, from the statement diagnostics area.

- `MESSAGE_TEXT`

  The statement error message, from the statement diagnostics area.

- `ERRORS`

  Whether an error occurred for the statement. The value is 0 if the SQLSTATE value begins with `00` (completion) or `01` (warning). The value is 1 is the SQLSTATE value is anything else.

- `WARNINGS`

  The number of warnings, from the statement diagnostics area.

- `ROWS_AFFECTED`

  The number of rows affected by the statement. For a description of the meaning of “affected,” see mysql_affected_rows().

- `ROWS_SENT`

  The number of rows returned by the statement.

- `ROWS_EXAMINED`

  The number of rows examined by the server layer (not counting any processing internal to storage engines).

- `CREATED_TMP_DISK_TABLES`

  Like the `Created_tmp_disk_tables` status variable, but specific to the statement.

- `CREATED_TMP_TABLES`

  Like the `Created_tmp_tables` status variable, but specific to the statement.

- `SELECT_FULL_JOIN`

  Like the `Select_full_join` status variable, but specific to the statement.

- `SELECT_FULL_RANGE_JOIN`

  Like the `Select_full_range_join` status variable, but specific to the statement.

- `SELECT_RANGE`

  Like the `Select_range` status variable, but specific to the statement.

- `SELECT_RANGE_CHECK`

Like the `Select_range_check` status variable, but specific to the statement.

- `SELECT_SCAN`

  Like the `Select_scan` status variable, but specific to the statement.

- `SORT_MERGE_PASSES`

  Like the `Sort_merge_passes` status variable, but specific to the statement.

- `SORT_RANGE`

  Like the `Sort_range` status variable, but specific to the statement.

- `SORT_ROWS`

  Like the `Sort_rows` status variable, but specific to the statement.

- `SORT_SCAN`

  Like the `Sort_scan` status variable, but specific to the statement.

- `NO_INDEX_USED`

  1 if the statement performed a table scan without using an index, 0 otherwise.

- `NO_GOOD_INDEX_USED`

  1 if the server found no good index to use for the statement, 0 otherwise. For additional information, see the description of the `Extra` column from `EXPLAIN` output for the `Range checked for each record` value in EXPLAIN Output Format.

- `NESTING_EVENT_ID`, `NESTING_EVENT_TYPE`, `NESTING_EVENT_LEVEL`

  These three columns are used with other columns to provide information as follows for top-level (unnested) statements and nested statements (executed within a stored program).

  For top level statements:

  ```
  OBJECT_TYPE = NULL
  OBJECT_SCHEMA = NULL
  OBJECT_NAME = NULL
  NESTING_EVENT_ID = the parent transaction EVENT_ID
  NESTING_EVENT_TYPE = 'TRANSACTION'
  NESTING_LEVEL = 0
  ```

  For nested statements:

  ```
  OBJECT_TYPE = the parent statement object type
  OBJECT_SCHEMA = the parent statement object schema
  OBJECT_NAME = the parent statement object name
  NESTING_EVENT_ID = the parent statement EVENT_ID
  NESTING_EVENT_TYPE = 'STATEMENT'
  NESTING_LEVEL = the parent statement NESTING_LEVEL plus one
  ```

- `STATEMENT_ID`

  The query ID maintained by the server at the SQL level. The value is unique for the server instance because these IDs are generated using a global counter that is incremented atomically. This column was added in MySQL 8.0.14.

- `CPU_TIME`

  The time spent on CPU for the current thread, expressed in picoseconds. This column was added in MySQL 8.0.28.

- `MAX_CONTROLLED_MEMORY`

  Reports the maximum amount of controlled memory used by a statement during execution.

  This column was added in MySQL 8.0.31.

- `MAX_TOTAL_MEMORY`

  Reports the maximum amount of memory used by a statement during execution.

  This column was added in MySQL 8.0.31.

- `EXECUTION_ENGINE`

  The query execution engine. The value is either `PRIMARY` or `SECONDARY`. For use with HeatWave Service and HeatWave, where the `PRIMARY` engine is `InnoDB` and the `SECONDARY` engine is HeatWave (`RAPID`). For MySQL Community Edition Server, MySQL Enterprise Edition Server (on-premise), and HeatWave Service without HeatWave, the value is always `PRIMARY`. This column was added in MySQL 8.0.29.

The `events_statements_current` table has these indexes:

- Primary key on (`THREAD_ID`, `EVENT_ID`)

`TRUNCATE TABLE` is permitted for the `events_statements_current` table. It removes the rows.

10.6.2 The events_statements_history Table

The `events_statements_history` table contains the *`N`* most recent statement events that have ended per thread. Statement events are not added to the table until they have ended. When the table contains the maximum number of rows for a given thread, the oldest thread row is discarded when a new row for that thread is added. When a thread ends, all its rows are discarded.

The Performance Schema autosizes the value of *`N`* during server startup. To set the number of rows per thread explicitly, set the `performance_schema_events_statements_history_size` system variable at server startup.

The `events_statements_history` table has the same columns and indexing as `events_statements_current`. See Section 10.6.1, “The events_statements_current Table”.

`TRUNCATE TABLE` is permitted for the `events_statements_history` table. It removes the rows.

For more information about the relationship between the three `events_statements_`*`xxx`* event tables, see Performance Schema Tables for Current and Historical Events.

For information about configuring whether to collect statement events, see Section 10.6, “Performance Schema Statement Event Tables”.

10.6.3 The events_statements_history_long Table

The `events_statements_history_long` table contains the *`N`* most recent statement events that have ended globally, across all threads. Statement events are not added to the table until they have ended.

When the table becomes full, the oldest row is discarded when a new row is added, regardless of which thread generated either row.

The value of *N* is autosized at server startup. To set the table size explicitly, set the `performance_schema_events_statements_history_long_size` system variable at server startup.

The `events_statements_history_long` table has the same columns as `events_statements_current`. See Section 10.6.1, "The events_statements_current Table". Unlike `events_statements_current`, `events_statements_history_long` has no indexing.

`TRUNCATE TABLE` is permitted for the `events_statements_history_long` table. It removes the rows.

For more information about the relationship between the three `events_statements_xxx` event tables, see Performance Schema Tables for Current and Historical Events.

For information about configuring whether to collect statement events, see Section 10.6, "Performance Schema Statement Event Tables".

10.6.4 The prepared_statements_instances Table

The Performance Schema provides instrumentation for prepared statements, for which there are two protocols:

- The binary protocol. This is accessed through the MySQL C API and maps onto underlying server commands as shown in the following table.

C API Function	Corresponding Server Command
`mysql_stmt_prepare()`	`COM_STMT_PREPARE`
`mysql_stmt_execute()`	`COM_STMT_EXECUTE`
`mysql_stmt_close()`	`COM_STMT_CLOSE`

- The text protocol. This is accessed using SQL statements and maps onto underlying server commands as shown in the following table.

SQL Statement	Corresponding Server Command
`PREPARE`	`SQLCOM_PREPARE`
`EXECUTE`	`SQLCOM_EXECUTE`
`DEALLOCATE PREPARE`, `DROP PREPARE`	`SQLCOM_DEALLOCATE PREPARE`

Performance Schema prepared statement instrumentation covers both protocols. The following discussion refers to the server commands rather than the C API functions or SQL statements.

Information about prepared statements is available in the `prepared_statements_instances` table. This table enables inspection of prepared statements used in the server and provides aggregated statistics about them. To control the size of this table, set the `performance_schema_max_prepared_statements_instances` system variable at server startup.

Collection of prepared statement information depends on the statement instruments shown in the following table. These instruments are enabled by default. To modify them, update the `setup_instruments` table.

Instrument	Server Command
`statement/com/Prepare`	`COM_STMT_PREPARE`

Instrument	Server Command
statement/com/Execute	COM_STMT_EXECUTE
statement/sql/prepare_sql	SQLCOM_PREPARE
statement/sql/execute_sql	SQLCOM_EXECUTE

The Performance Schema manages the contents of the prepared_statements_instances table as follows:

- Statement preparation

  A COM_STMT_PREPARE or SQLCOM_PREPARE command creates a prepared statement in the server. If the statement is successfully instrumented, a new row is added to the prepared_statements_instances table. If the statement cannot be instrumented, Performance_schema_prepared_statements_lost status variable is incremented.

- Prepared statement execution

  Execution of a COM_STMT_EXECUTE or SQLCOM_PREPARE command for an instrumented prepared statement instance updates the corresponding prepared_statements_instances table row.

- Prepared statement deallocation

  Execution of a COM_STMT_CLOSE or SQLCOM_DEALLOCATE_PREPARE command for an instrumented prepared statement instance removes the corresponding prepared_statements_instances table row. To avoid resource leaks, removal occurs even if the prepared statement instruments described previously are disabled.

The prepared_statements_instances table has these columns:

- OBJECT_INSTANCE_BEGIN

  The address in memory of the instrumented prepared statement.

- STATEMENT_ID

  The internal statement ID assigned by the server. The text and binary protocols both use statement IDs.

- STATEMENT_NAME

  For the binary protocol, this column is NULL. For the text protocol, this column is the external statement name assigned by the user. For example, for the following SQL statement, the name of the prepared statement is stmt:

  ```
  PREPARE stmt FROM 'SELECT 1';
  ```

- SQL_TEXT

  The prepared statement text, with ? placeholder markers.

- OWNER_THREAD_ID, OWNER_EVENT_ID

  These columns indicate the event that created the prepared statement.

- OWNER_OBJECT_TYPE, OWNER_OBJECT_SCHEMA, OWNER_OBJECT_NAME

  For a prepared statement created by a client session, these columns are NULL. For a prepared statement created by a stored program, these columns point to the stored program. A typical user error

is forgetting to deallocate prepared statements. These columns can be used to find stored programs that leak prepared statements:

```
SELECT
  OWNER_OBJECT_TYPE, OWNER_OBJECT_SCHEMA, OWNER_OBJECT_NAME,
  STATEMENT_NAME, SQL_TEXT
FROM performance_schema.prepared_statements_instances
WHERE OWNER_OBJECT_TYPE IS NOT NULL;
```

- The query execution engine. The value is either `PRIMARY` or `SECONDARY`. For use with HeatWave Service and HeatWave, where the `PRIMARY` engine is `InnoDB` and the `SECONDARY` engine is HeatWave (`RAPID`). For MySQL Community Edition Server, MySQL Enterprise Edition Server (on-premise), and HeatWave Service without HeatWave, the value is always `PRIMARY`. This column was added in MySQL 8.0.29.

- `TIMER_PREPARE`

  The time spent executing the statement preparation itself.

- `COUNT_REPREPARE`

  The number of times the statement was reprepared internally (see Caching of Prepared Statements and Stored Programs). Timing statistics for repreparation are not available because it is counted as part of statement execution, not as a separate operation.

- `COUNT_EXECUTE`, `SUM_TIMER_EXECUTE`, `MIN_TIMER_EXECUTE`, `AVG_TIMER_EXECUTE`, `MAX_TIMER_EXECUTE`

  Aggregated statistics for executions of the prepared statement.

- `SUM_`*`xxx`*

  The remaining `SUM_`*`xxx`* columns are the same as for the statement summary tables (see Section 10.20.3, “Statement Summary Tables”).

- `MAX_CONTROLLED_MEMORY`

  Reports the maximum amount of controlled memory used by a prepared statement during execution.

  This column was added in MySQL 8.0.31.

- `MAX_TOTAL_MEMORY`

  Reports the maximum amount of memory used by a prepared statement during execution.

  This column was added in MySQL 8.0.31.

The `prepared_statements_instances` table has these indexes:

- Primary key on (`OBJECT_INSTANCE_BEGIN`)
- Index on (`STATEMENT_ID`)
- Index on (`STATEMENT_NAME`)
- Index on (`OWNER_THREAD_ID`, `OWNER_EVENT_ID`)
- Index on (`OWNER_OBJECT_TYPE`, `OWNER_OBJECT_SCHEMA`, `OWNER_OBJECT_NAME`)

`TRUNCATE TABLE` resets the statistics columns of the `prepared_statements_instances` table.

10.7 Performance Schema Transaction Tables

The Performance Schema instruments transactions. Within the event hierarchy, wait events nest within stage events, which nest within statement events, which nest within transaction events.

These tables store transaction events:

- `events_transactions_current`: The current transaction event for each thread.
- `events_transactions_history`: The most recent transaction events that have ended per thread.
- `events_transactions_history_long`: The most recent transaction events that have ended globally (across all threads).

The following sections describe the transaction event tables. There are also summary tables that aggregate information about transaction events; see Section 10.20.5, “Transaction Summary Tables”.

For more information about the relationship between the three transaction event tables, see Performance Schema Tables for Current and Historical Events.

- Configuring Transaction Event Collection
- Transaction Boundaries
- Transaction Instrumentation
- Transactions and Nested Events
- Transactions and Stored Programs
- Transactions and Savepoints
- Transactions and Errors

Configuring Transaction Event Collection

To control whether to collect transaction events, set the state of the relevant instruments and consumers:

- The `setup_instruments` table contains an instrument named `transaction`. Use this instrument to enable or disable collection of individual transaction event classes.
- The `setup_consumers` table contains consumer values with names corresponding to the current and historical transaction event table names. Use these consumers to filter collection of transaction events.

The `transaction` instrument and the `events_transactions_current` and `events_transactions_history` transaction consumers are enabled by default:

```
mysql> SELECT NAME, ENABLED, TIMED
       FROM performance_schema.setup_instruments
       WHERE NAME = 'transaction';
+-------------+---------+-------+
| NAME        | ENABLED | TIMED |
+-------------+---------+-------+
| transaction | YES     | YES   |
+-------------+---------+-------+
mysql> SELECT *
       FROM performance_schema.setup_consumers
       WHERE NAME LIKE 'events_transactions%';
+--------------------------------+---------+
| NAME                           | ENABLED |
```

```
+---------------------------------+---------+
| events_transactions_current     | YES     |
| events_transactions_history     | YES     |
| events_transactions_history_long | NO      |
+---------------------------------+---------+
```

To control transaction event collection at server startup, use lines like these in your `my.cnf` file:

- Enable:

```
[mysqld]
performance-schema-instrument='transaction=ON'
performance-schema-consumer-events-transactions-current=ON
performance-schema-consumer-events-transactions-history=ON
performance-schema-consumer-events-transactions-history-long=ON
```

- Disable:

```
[mysqld]
performance-schema-instrument='transaction=OFF'
performance-schema-consumer-events-transactions-current=OFF
performance-schema-consumer-events-transactions-history=OFF
performance-schema-consumer-events-transactions-history-long=OFF
```

To control transaction event collection at runtime, update the `setup_instruments` and `setup_consumers` tables:

- Enable:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'YES', TIMED = 'YES'
WHERE NAME = 'transaction';
UPDATE performance_schema.setup_consumers
SET ENABLED = 'YES'
WHERE NAME LIKE 'events_transactions%';
```

- Disable:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'NO', TIMED = 'NO'
WHERE NAME = 'transaction';
UPDATE performance_schema.setup_consumers
SET ENABLED = 'NO'
WHERE NAME LIKE 'events_transactions%';
```

To collect transaction events only for specific transaction event tables, enable the `transaction` instrument but only the transaction consumers corresponding to the desired tables.

For additional information about configuring event collection, see Chapter 4, *Performance Schema Startup Configuration*, and Chapter 5, *Performance Schema Runtime Configuration*.

Transaction Boundaries

In MySQL Server, transactions start explicitly with these statements:

```
START TRANSACTION | BEGIN | XA START | XA BEGIN
```

Transactions also start implicitly. For example, when the `autocommit` system variable is enabled, the start of each statement starts a new transaction.

When `autocommit` is disabled, the first statement following a committed transaction marks the start of a new transaction. Subsequent statements are part of the transaction until it is committed.

Transactions explicitly end with these statements:

```
COMMIT | ROLLBACK | XA COMMIT | XA ROLLBACK
```

Transactions also end implicitly, by execution of DDL statements, locking statements, and server administration statements.

In the following discussion, references to `START TRANSACTION` also apply to `BEGIN`, `XA START`, and `XA BEGIN`. Similarly, references to `COMMIT` and `ROLLBACK` apply to `XA COMMIT` and `XA ROLLBACK`, respectively.

The Performance Schema defines transaction boundaries similarly to that of the server. The start and end of a transaction event closely match the corresponding state transitions in the server:

- For an explicitly started transaction, the transaction event starts during processing of the `START TRANSACTION` statement.
- For an implicitly started transaction, the transaction event starts on the first statement that uses a transactional engine after the previous transaction has ended.
- For any transaction, whether explicitly or implicitly ended, the transaction event ends when the server transitions out of the active transaction state during the processing of `COMMIT` or `ROLLBACK`.

There are subtle implications to this approach:

- Transaction events in the Performance Schema do not fully include the statement events associated with the corresponding `START TRANSACTION`, `COMMIT`, or `ROLLBACK` statements. There is a trivial amount of timing overlap between the transaction event and these statements.
- Statements that work with nontransactional engines have no effect on the transaction state of the connection. For implicit transactions, the transaction event begins with the first statement that uses a transactional engine. This means that statements operating exclusively on nontransactional tables are ignored, even following `START TRANSACTION`.

To illustrate, consider the following scenario:

```
1. SET autocommit = OFF;
2. CREATE TABLE t1 (a INT) ENGINE = InnoDB;
3. START TRANSACTION;                       -- Transaction 1 START
4. INSERT INTO t1 VALUES (1), (2), (3);
5. CREATE TABLE t2 (a INT) ENGINE = MyISAM; -- Transaction 1 COMMIT
                                            -- (implicit; DDL forces commit)
6. INSERT INTO t2 VALUES (1), (2), (3);     -- Update nontransactional table
7. UPDATE t2 SET a = a + 1;                 -- ... and again
8. INSERT INTO t1 VALUES (4), (5), (6);     -- Write to transactional table
                                            -- Transaction 2 START (implicit)
9. COMMIT;                                  -- Transaction 2 COMMIT
```

From the perspective of the server, Transaction 1 ends when table `t2` is created. Transaction 2 does not start until a transactional table is accessed, despite the intervening updates to nontransactional tables.

From the perspective of the Performance Schema, Transaction 2 starts when the server transitions into an active transaction state. Statements 6 and 7 are not included within the boundaries of Transaction 2, which is consistent with how the server writes transactions to the binary log.

Transaction Instrumentation

Three attributes define transactions:

- Access mode (read only, read write)
- Isolation level (`SERIALIZABLE`, `REPEATABLE READ`, and so forth)
- Implicit (`autocommit` enabled) or explicit (`autocommit` disabled)

To reduce complexity of the transaction instrumentation and to ensure that the collected transaction data provides complete, meaningful results, all transactions are instrumented independently of access mode, isolation level, or autocommit mode.

To selectively examine transaction history, use the attribute columns in the transaction event tables: `ACCESS_MODE`, `ISOLATION_LEVEL`, and `AUTOCOMMIT`.

The cost of transaction instrumentation can be reduced various ways, such as enabling or disabling transaction instrumentation according to user, account, host, or thread (client connection).

Transactions and Nested Events

The parent of a transaction event is the event that initiated the transaction. For an explicitly started transaction, this includes the `START TRANSACTION` and `COMMIT AND CHAIN` statements. For an implicitly started transaction, it is the first statement that uses a transactional engine after the previous transaction ends.

In general, a transaction is the top-level parent to all events initiated during the transaction, including statements that explicitly end the transaction such as `COMMIT` and `ROLLBACK`. Exceptions are statements that implicitly end a transaction, such as DDL statements, in which case the current transaction must be committed before the new statement is executed.

Transactions and Stored Programs

Transactions and stored program events are related as follows:

- Stored Procedures

  Stored procedures operate independently of transactions. A stored procedure can be started within a transaction, and a transaction can be started or ended from within a stored procedure. If called from within a transaction, a stored procedure can execute statements that force a commit of the parent transaction and then start a new transaction.

  If a stored procedure is started within a transaction, that transaction is the parent of the stored procedure event.

  If a transaction is started by a stored procedure, the stored procedure is the parent of the transaction event.

- Stored Functions

  Stored functions are restricted from causing an explicit or implicit commit or rollback. Stored function events can reside within a parent transaction event.

- Triggers

  Triggers activate as part of a statement that accesses the table with which it is associated, so the parent of a trigger event is always the statement that activates it.

  Triggers cannot issue statements that cause an explicit or implicit commit or rollback of a transaction.

- Scheduled Events

  The execution of the statements in the body of a scheduled event takes place in a new connection. Nesting of a scheduled event within a parent transaction is not applicable.

Transactions and Savepoints

Savepoint statements are recorded as separate statement events. Transaction events include separate counters for `SAVEPOINT`, `ROLLBACK TO SAVEPOINT`, and `RELEASE SAVEPOINT` statements issued during the transaction.

Transactions and Errors

Errors and warnings that occur within a transaction are recorded in statement events, but not in the corresponding transaction event. This includes transaction-specific errors and warnings, such as a rollback on a nontransactional table or GTID consistency errors.

10.7.1 The events_transactions_current Table

The `events_transactions_current` table contains current transaction events. The table stores one row per thread showing the current status of the thread's most recent monitored transaction event, so there is no system variable for configuring the table size. For example:

```
mysql> SELECT *
       FROM performance_schema.events_transactions_current LIMIT 1\G
*************************** 1. row ***************************
                    THREAD_ID: 26
                     EVENT_ID: 7
                 END_EVENT_ID: NULL
                   EVENT_NAME: transaction
                        STATE: ACTIVE
                       TRX_ID: NULL
                         GTID: 3E11FA47-71CA-11E1-9E33-C80AA9429562:56
                          XID: NULL
                     XA_STATE: NULL
                       SOURCE: transaction.cc:150
                  TIMER_START: 420833537900000
                    TIMER_END: NULL
                   TIMER_WAIT: NULL
                  ACCESS_MODE: READ WRITE
              ISOLATION_LEVEL: REPEATABLE READ
                   AUTOCOMMIT: NO
           NUMBER_OF_SAVEPOINTS: 0
NUMBER_OF_ROLLBACK_TO_SAVEPOINT: 0
    NUMBER_OF_RELEASE_SAVEPOINT: 0
         OBJECT_INSTANCE_BEGIN: NULL
              NESTING_EVENT_ID: 6
            NESTING_EVENT_TYPE: STATEMENT
```

Of the tables that contain transaction event rows, `events_transactions_current` is the most fundamental. Other tables that contain transaction event rows are logically derived from the current events. For example, the `events_transactions_history` and `events_transactions_history_long` tables are collections of the most recent transaction events that have ended, up to a maximum number of rows per thread and globally across all threads, respectively.

For more information about the relationship between the three transaction event tables, see Performance Schema Tables for Current and Historical Events.

For information about configuring whether to collect transaction events, see Section 10.7, “Performance Schema Transaction Tables”.

The `events_transactions_current` table has these columns:

- `THREAD_ID`, `EVENT_ID`

  The thread associated with the event and the thread current event number when the event starts. The `THREAD_ID` and `EVENT_ID` values taken together uniquely identify the row. No two rows have the same pair of values.

- `END_EVENT_ID`

  This column is set to `NULL` when the event starts and updated to the thread current event number when the event ends.

- `EVENT_NAME`

  The name of the instrument from which the event was collected. This is a `NAME` value from the `setup_instruments` table. Instrument names may have multiple parts and form a hierarchy, as discussed in Chapter 7, *Performance Schema Instrument Naming Conventions*.

- `STATE`

  The current transaction state. The value is `ACTIVE` (after `START TRANSACTION` or `BEGIN`), `COMMITTED` (after `COMMIT`), or `ROLLED BACK` (after `ROLLBACK`).

- `TRX_ID`

  Unused.

- `GTID`

  The GTID column contains the value of `gtid_next`, which can be one of `ANONYMOUS`, `AUTOMATIC`, or a GTID using the format `UUID:NUMBER`. For transactions that use `gtid_next=AUTOMATIC`, which is all normal client transactions, the GTID column changes when the transaction commits and the actual GTID is assigned. If `gtid_mode` is either `ON` or `ON_PERMISSIVE`, the GTID column changes to the transaction's GTID. If `gtid_mode` is either `OFF` or `OFF_PERMISSIVE`, the GTID column changes to `ANONYMOUS`.

- `XID_FORMAT_ID`, `XID_GTRID`, and `XID_BQUAL`

  The elements of the XA transaction identifier. They have the format described in XA Transaction SQL Statements.

- `XA_STATE`

  The state of the XA transaction. The value is `ACTIVE` (after `XA START`), `IDLE` (after `XA END`), `PREPARED` (after `XA PREPARE`), `ROLLED BACK` (after `XA ROLLBACK`), or `COMMITTED` (after `XA COMMIT`).

  On a replica, the same XA transaction can appear in the `events_transactions_current` table with different states on different threads. This is because immediately after the XA transaction is prepared, it is detached from the replica's applier thread, and can be committed or rolled back by any thread on the replica. The `events_transactions_current` table displays the current status of the most recent monitored transaction event on the thread, and does not update this status when the thread is idle. So the XA transaction can still be displayed in the `PREPARED` state for the original applier thread, after it has been processed by another thread. To positively identify XA transactions that are still in the `PREPARED` state and need to be recovered, use the `XA RECOVER` statement rather than the Performance Schema transaction tables.

- SOURCE

  The name of the source file containing the instrumented code that produced the event and the line number in the file at which the instrumentation occurs. This enables you to check the source to determine exactly what code is involved.

- TIMER_START, TIMER_END, TIMER_WAIT

  Timing information for the event. The unit for these values is picoseconds (trillionths of a second). The TIMER_START and TIMER_END values indicate when event timing started and ended. TIMER_WAIT is the event elapsed time (duration).

  If an event has not finished, TIMER_END is the current timer value and TIMER_WAIT is the time elapsed so far (TIMER_END − TIMER_START).

  If an event is produced from an instrument that has TIMED = NO, timing information is not collected, and TIMER_START, TIMER_END, and TIMER_WAIT are all NULL.

  For discussion of picoseconds as the unit for event times and factors that affect time values, see Section 5.1, “Performance Schema Event Timing”.

- ACCESS_MODE

  The transaction access mode. The value is READ WRITE or READ ONLY.

- ISOLATION_LEVEL

  The transaction isolation level. The value is REPEATABLE READ, READ COMMITTED, READ UNCOMMITTED, or SERIALIZABLE.

- AUTOCOMMIT

  Whether autocommit mode was enabled when the transaction started.

- NUMBER_OF_SAVEPOINTS, NUMBER_OF_ROLLBACK_TO_SAVEPOINT, NUMBER_OF_RELEASE_SAVEPOINT

  The number of SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT statements issued during the transaction.

- OBJECT_INSTANCE_BEGIN

  Unused.

- NESTING_EVENT_ID

  The EVENT_ID value of the event within which this event is nested.

- NESTING_EVENT_TYPE

  The nesting event type. The value is TRANSACTION, STATEMENT, STAGE, or WAIT. (TRANSACTION does not appear because transactions cannot be nested.)

The events_transactions_current table has these indexes:

- Primary key on (THREAD_ID, EVENT_ID)

TRUNCATE TABLE is permitted for the events_transactions_current table. It removes the rows.

10.7.2 The events_transactions_history Table

The `events_transactions_history` table contains the *N* most recent transaction events that have ended per thread. Transaction events are not added to the table until they have ended. When the table contains the maximum number of rows for a given thread, the oldest thread row is discarded when a new row for that thread is added. When a thread ends, all its rows are discarded.

The Performance Schema autosizes the value of *N* during server startup. To set the number of rows per thread explicitly, set the `performance_schema_events_transactions_history_size` system variable at server startup.

The `events_transactions_history` table has the same columns and indexing as `events_transactions_current`. See Section 10.7.1, “The events_transactions_current Table”.

`TRUNCATE TABLE` is permitted for the `events_transactions_history` table. It removes the rows.

For more information about the relationship between the three transaction event tables, see Performance Schema Tables for Current and Historical Events.

For information about configuring whether to collect transaction events, see Section 10.7, “Performance Schema Transaction Tables”.

10.7.3 The events_transactions_history_long Table

The `events_transactions_history_long` table contains the *N* most recent transaction events that have ended globally, across all threads. Transaction events are not added to the table until they have ended. When the table becomes full, the oldest row is discarded when a new row is added, regardless of which thread generated either row.

The Performance Schema autosizes the value of *N* is autosized at server startup. To set the table size explicitly, set the `performance_schema_events_transactions_history_long_size` system variable at server startup.

The `events_transactions_history_long` table has the same columns as `events_transactions_current`. See Section 10.7.1, “The events_transactions_current Table”. Unlike `events_transactions_current`, `events_transactions_history_long` has no indexing.

`TRUNCATE TABLE` is permitted for the `events_transactions_history_long` table. It removes the rows.

For more information about the relationship between the three transaction event tables, see Performance Schema Tables for Current and Historical Events.

For information about configuring whether to collect transaction events, see Section 10.7, “Performance Schema Transaction Tables”.

10.8 Performance Schema Connection Tables

When a client connects to the MySQL server, it does so under a particular user name and from a particular host. The Performance Schema provides statistics about these connections, tracking them per account (user and host combination) as well as separately per user name and host name, using these tables:

- `accounts`: Connection statistics per client account
- `hosts`: Connection statistics per client host name
- `users`: Connection statistics per client user name

The meaning of “account” in the connection tables is similar to its meaning in the MySQL grant tables in the `mysql` system database, in the sense that the term refers to a combination of user and host values. They differ in that, for grant tables, the host part of an account can be a pattern, whereas for Performance Schema tables, the host value is always a specific nonpattern host name.

Each connection table has `CURRENT_CONNECTIONS` and `TOTAL_CONNECTIONS` columns to track the current and total number of connections per “tracking value” on which its statistics are based. The tables differ in what they use for the tracking value. The `accounts` table has `USER` and `HOST` columns to track connections per user and host combination. The `users` and `hosts` tables have a `USER` and `HOST` column, respectively, to track connections per user name and host name.

The Performance Schema also counts internal threads and threads for user sessions that failed to authenticate, using rows with `USER` and `HOST` column values of `NULL`.

Suppose that clients named `user1` and `user2` each connect one time from `hosta` and `hostb`. The Performance Schema tracks the connections as follows:

- The `accounts` table has four rows, for the `user1`/`hosta`, `user1`/`hostb`, `user2`/`hosta`, and `user2`/`hostb` account values, each row counting one connection per account.
- The `hosts` table has two rows, for `hosta` and `hostb`, each row counting two connections per host name.
- The `users` table has two rows, for `user1` and `user2`, each row counting two connections per user name.

When a client connects, the Performance Schema determines which row in each connection table applies, using the tracking value appropriate to each table. If there is no such row, one is added. Then the Performance Schema increments by one the `CURRENT_CONNECTIONS` and `TOTAL_CONNECTIONS` columns in that row.

When a client disconnects, the Performance Schema decrements by one the `CURRENT_CONNECTIONS` column in the row and leaves the `TOTAL_CONNECTIONS` column unchanged.

`TRUNCATE TABLE` is permitted for connection tables. It has these effects:

- Rows are removed for accounts, hosts, or users that have no current connections (rows with `CURRENT_CONNECTIONS = 0`).
- Nonremoved rows are reset to count only current connections: For rows with `CURRENT_CONNECTIONS > 0`, `TOTAL_CONNECTIONS` is reset to `CURRENT_CONNECTIONS`.
- Summary tables that depend on the connection table are implicitly truncated, as described later in this section.

The Performance Schema maintains summary tables that aggregate connection statistics for various event types by account, host, or user. These tables have `_summary_by_account`, `_summary_by_host`, or `_summary_by_user` in the name. To identify them, use this query:

```
mysql> SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
       WHERE TABLE_SCHEMA = 'performance_schema'
       AND TABLE_NAME REGEXP '_summary_by_(account|host|user)'
       ORDER BY TABLE_NAME;
+-----------------------------------------------------+
| TABLE_NAME                                          |
+-----------------------------------------------------+
| events_errors_summary_by_account_by_error           |
| events_errors_summary_by_host_by_error              |
| events_errors_summary_by_user_by_error              |
```

```
| events_stages_summary_by_account_by_event_name       |
| events_stages_summary_by_host_by_event_name          |
| events_stages_summary_by_user_by_event_name          |
| events_statements_summary_by_account_by_event_name   |
| events_statements_summary_by_host_by_event_name      |
| events_statements_summary_by_user_by_event_name      |
| events_transactions_summary_by_account_by_event_name |
| events_transactions_summary_by_host_by_event_name    |
| events_transactions_summary_by_user_by_event_name    |
| events_waits_summary_by_account_by_event_name        |
| events_waits_summary_by_host_by_event_name           |
| events_waits_summary_by_user_by_event_name           |
| memory_summary_by_account_by_event_name              |
| memory_summary_by_host_by_event_name                 |
| memory_summary_by_user_by_event_name                 |
+------------------------------------------------------+
```

For details about individual connection summary tables, consult the section that describes tables for the summarized event type:

- Wait event summaries: Section 10.20.1, “Wait Event Summary Tables”
- Stage event summaries: Section 10.20.2, “Stage Summary Tables”
- Statement event summaries: Section 10.20.3, “Statement Summary Tables”
- Transaction event summaries: Section 10.20.5, “Transaction Summary Tables”
- Memory event summaries: Section 10.20.10, “Memory Summary Tables”
- Error event summaries: Section 10.20.11, “Error Summary Tables”

`TRUNCATE TABLE` is permitted for connection summary tables. It removes rows for accounts, hosts, or users with no connections, and resets the summary columns to zero for the remaining rows. In addition, each summary table that is aggregated by account, host, user, or thread is implicitly truncated by truncation of the connection table on which it depends. The following table describes the relationship between connection table truncation and implicitly truncated tables.

Table 10.2 Implicit Effects of Connection Table Truncation

Truncated Connection Table	Implicitly Truncated Summary Tables
`accounts`	Tables with names containing `_summary_by_account`, `_summary_by_thread`
`hosts`	Tables with names containing `_summary_by_account`, `_summary_by_host`, `_summary_by_thread`
`users`	Tables with names containing `_summary_by_account`, `_summary_by_user`, `_summary_by_thread`

Truncating a `_summary_global` summary table also implicitly truncates its corresponding connection and thread summary tables. For example, truncating `events_waits_summary_global_by_event_name` implicitly truncates the wait event summary tables that are aggregated by account, host, user, or thread.

10.8.1 The accounts Table

The `accounts` table contains a row for each account that has connected to the MySQL server. For each account, the table counts the current and total number of connections. The table size is autosized at server

startup. To set the table size explicitly, set the `performance_schema_accounts_size` system variable at server startup. To disable account statistics, set this variable to 0.

The `accounts` table has the following columns. For a description of how the Performance Schema maintains rows in this table, including the effect of `TRUNCATE TABLE`, see Section 10.8, "Performance Schema Connection Tables".

- `USER`

  The client user name for the connection. This is `NULL` for an internal thread, or for a user session that failed to authenticate.

- `HOST`

  The host from which the client connected. This is `NULL` for an internal thread, or for a user session that failed to authenticate.

- `CURRENT_CONNECTIONS`

  The current number of connections for the account.

- `TOTAL_CONNECTIONS`

  The total number of connections for the account.

- `MAX_SESSION_CONTROLLED_MEMORY`

  Reports the maximum amount of controlled memory used by a session belonging to the account.

  This column was added in MySQL 8.0.31.

- `MAX_SESSION_TOTAL_MEMORY`

  Reports the maximum amount of memory used by a session belonging to the account.

  This column was added in MySQL 8.0.31.

The `accounts` table has these indexes:

- Primary key on (`USER`, `HOST`)

10.8.2 The hosts Table

The `hosts` table contains a row for each host from which clients have connected to the MySQL server. For each host name, the table counts the current and total number of connections. The table size is autosized at server startup. To set the table size explicitly, set the `performance_schema_hosts_size` system variable at server startup. To disable host statistics, set this variable to 0.

The `hosts` table has the following columns. For a description of how the Performance Schema maintains rows in this table, including the effect of `TRUNCATE TABLE`, see Section 10.8, "Performance Schema Connection Tables".

- `HOST`

  The host from which the client connected. This is `NULL` for an internal thread, or for a user session that failed to authenticate.

- `CURRENT_CONNECTIONS`

The current number of connections for the host.

- `TOTAL_CONNECTIONS`

  The total number of connections for the host.

- `MAX_SESSION_CONTROLLED_MEMORY`

  Reports the maximum amount of controlled memory used by a session belonging to the host.

  This column was added in MySQL 8.0.31.

- `MAX_SESSION_TOTAL_MEMORY`

  Reports the maximum amount of memory used by a session belonging to the host.

  This column was added in MySQL 8.0.31.

The `hosts` table has these indexes:

- Primary key on (`HOST`)

10.8.3 The users Table

The `users` table contains a row for each user who has connected to the MySQL server. For each user name, the table counts the current and total number of connections. The table size is autosized at server startup. To set the table size explicitly, set the `performance_schema_users_size` system variable at server startup. To disable user statistics, set this variable to 0.

The `users` table has the following columns. For a description of how the Performance Schema maintains rows in this table, including the effect of `TRUNCATE TABLE`, see Section 10.8, “Performance Schema Connection Tables”.

- `USER`

  The client user name for the connection. This is `NULL` for an internal thread, or for a user session that failed to authenticate.

- `CURRENT_CONNECTIONS`

  The current number of connections for the user.

- `TOTAL_CONNECTIONS`

  The total number of connections for the user.

- `MAX_SESSION_CONTROLLED_MEMORY`

  Reports the maximum amount of controlled memory used by a session belonging to the user.

  This column was added in MySQL 8.0.31.

- `MAX_SESSION_TOTAL_MEMORY`

  Reports the maximum amount of memory used by a session belonging to the user.

  This column was added in MySQL 8.0.31.

The `users` table has these indexes:

- Primary key on (`USER`)

10.9 Performance Schema Connection Attribute Tables

Connection attributes are key-value pairs that application programs can pass to the server at connect time. For applications based on the C API implemented by the `libmysqlclient` client library, the `mysql_options()` and `mysql_options4()` functions define the connection attribute set. Other MySQL Connectors may provide their own attribute-definition methods.

These Performance Schema tables expose attribute information:

- `session_account_connect_attrs`: Connection attributes for the current session, and other sessions associated with the session account
- `session_connect_attrs`: Connection attributes for all sessions

In addition, connect events written to the audit log may include connection attributes. See Audit Log File Formats.

Attribute names that begin with an underscore (_) are reserved for internal use and should not be created by application programs. This convention permits new attributes to be introduced by MySQL without colliding with application attributes, and enables application programs to define their own attributes that do not collide with internal attributes.

- Available Connection Attributes
- Connection Attribute Limits

Available Connection Attributes

The set of connection attributes visible within a given connection varies depending on factors such as your platform, MySQL Connector used to establish the connection, or client program.

The `libmysqlclient` client library sets these attributes:

- `_client_name`: The client name (`libmysql` for the client library).
- `_client_version`: The client library version.
- `_os`: The operating system (for example, `Linux`, `Win64`).
- `_pid`: The client process ID.
- `_platform`: The machine platform (for example, `x86_64`).
- `_thread`: The client thread ID (Windows only).

Other MySQL Connectors may define their own connection attributes.

MySQL Connector/C++ 8.0.16 and higher defines these attributes for applications that use X DevAPI or X DevAPI for C:

- `_client_license`: The connector license (for example `GPL-2.0`).
- `_client_name`: The connector name (`mysql-connector-cpp`).
- `_client_version`: The connector version.

- `_os`: The operating system (for example, `Linux`, `Win64`).
- `_pid`: The client process ID.
- `_platform`: The machine platform (for example, `x86_64`).
- `_source_host`: The host name of the machine on which the client is running.
- `_thread`: The client thread ID (Windows only).

MySQL Connector/J defines these attributes:

- `_client_name`: The client name
- `_client_version`: The client library version
- `_os`: The operating system (for example, `Linux`, `Win64`)
- `_client_license`: The connector license type
- `_platform`: The machine platform (for example, `x86_64`)
- `_runtime_vendor`: The Java runtime environment (JRE) vendor
- `_runtime_version`: The Java runtime environment (JRE) version

MySQL Connector/NET defines these attributes:

- `_client_version`: The client library version.
- `_os`: The operating system (for example, `Linux`, `Win64`).
- `_pid`: The client process ID.
- `_platform`: The machine platform (for example, `x86_64`).
- `_program_name`: The client name.
- `_thread`: The client thread ID (Windows only).

The Connector/Python 8.0.17 and higher implementation defines these attributes; some values and attributes depend on the Connector/Python implementation (pure python or c-ext):

- `_client_license`: The license type of the connector; `GPL-2.0` or `Commercial`. (pure python only)
- `_client_name`: Set to `mysql-connector-python` (pure python) or `libmysql` (c-ext)
- `_client_version`: The connector version (pure python) or mysqlclient library version (c-ext).
- `_os`: The operating system with the connector (for example, `Linux`, `Win64`).
- `_pid`: The process identifier on the source machine (for example, `26955`)
- `_platform`: The machine platform (for example, `x86_64`).
- `_source_host`: The host name of the machine on which the connector is connecting from.
- `_connector_version`: The connector version (for example, `8.0.42`) (c-ext only).
- `_connector_license`: The license type of the connector; `GPL-2.0` or `Commercial` (c-ext only).
- `_connector_name`: Always set to `mysql-connector-python` (c-ext only).

PHP defines attributes that depend on how it was compiled:

- Compiled using `libmysqlclient`: The standard `libmysqlclient` attributes, described previously.
- Compiled using `mysqlnd`: Only the `_client_name` attribute, with a value of `mysqlnd`.

Many MySQL client programs set a `program_name` attribute with a value equal to the client name. For example, `mysqladmin` and `mysqldump` set `program_name` to `mysqladmin` and `mysqldump`, respectively. MySQL Shell sets `program_name` to `mysqlsh`.

Some MySQL client programs define additional attributes:

- `mysql` (as of MySQL 8.0.17):
  - `os_user`: The name of the operating system user running the program. Available on Unix and Unix-like systems and Windows.
  - `os_sudouser`: The value of the `SUDO_USER` environment variable. Available on Unix and Unix-like systems.

  `mysql` connection attributes for which the value is empty are not sent.
- `mysqlbinlog`:
  - `_client_role`: `binary_log_listener`
- Replica connections:
  - `program_name`: `mysqld`
  - `_client_role`: `binary_log_listener`
  - `_client_replication_channel_name`: The channel name.
- `FEDERATED` storage engine connections:
  - `program_name`: `mysqld`
  - `_client_role`: `federated_storage`

Connection Attribute Limits

There are limits on the amount of connection attribute data transmitted from client to server:

- A fixed limit imposed by the client prior to connect time.
- A fixed limit imposed by the server at connect time.
- A configurable limit imposed by the Performance Schema at connect time.

For connections initiated using the C API, the `libmysqlclient` library imposes a limit of 64KB on the aggregate size of connection attribute data on the client side: Calls to `mysql_options()` that cause this limit to be exceeded produce a `CR_INVALID_PARAMETER_NO` error. Other MySQL Connectors may impose their own client-side limits on how much connection attribute data can be transmitted to the server.

On the server side, these size checks on connection attribute data occur:

- The server imposes a limit of 64KB on the aggregate size of connection attribute data it accepts. If a client attempts to send more than 64KB of attribute data, the server rejects the connection. Otherwise,

the server considers the attribute buffer valid and tracks the size of the longest such buffer in the `Performance_schema_session_connect_attrs_longest_seen` status variable.

- For accepted connections, the Performance Schema checks aggregate attribute size against the value of the `performance_schema_session_connect_attrs_size` system variable. If attribute size exceeds this value, these actions take place:
  - The Performance Schema truncates the attribute data and increments the `Performance_schema_session_connect_attrs_lost` status variable, which indicates the number of connections for which attribute truncation occurred.
  - The Performance Schema writes a message to the error log if the `log_error_verbosity` system variable is greater than 1:

    ```
    Connection attributes of length N were truncated
    (N bytes lost)
    for connection N, user user_name@host_name
    (as user_name), auth: {yes|no}
    ```

    The information in the warning message is intended to help DBAs identify clients for which attribute truncation occurred.
  - A `_truncated` attribute is added to the session attributes with a value indicating how many bytes were lost, if the attribute buffer has sufficient space. This enables the Performance Schema to expose per-connection truncation information in the connection attribute tables. This information can be examined without having to check the error log.

10.9.1 The session_account_connect_attrs Table

Application programs can provide key-value connection attributes to be passed to the server at connect time. For descriptions of common attributes, see Section 10.9, “Performance Schema Connection Attribute Tables”.

The `session_account_connect_attrs` table contains connection attributes only for the current session, and other sessions associated with the session account. To see connection attributes for all sessions, use the `session_connect_attrs` table.

The `session_account_connect_attrs` table has these columns:

- `PROCESSLIST_ID`

  The connection identifier for the session.

- `ATTR_NAME`

  The attribute name.

- `ATTR_VALUE`

  The attribute value.

- `ORDINAL_POSITION`

  The order in which the attribute was added to the set of connection attributes.

The `session_account_connect_attrs` table has these indexes:

- Primary key on (`PROCESSLIST_ID`, `ATTR_NAME`)

`TRUNCATE TABLE` is not permitted for the `session_account_connect_attrs` table.

10.9.2 The session_connect_attrs Table

Application programs can provide key-value connection attributes to be passed to the server at connect time. For descriptions of common attributes, see Section 10.9, “Performance Schema Connection Attribute Tables”.

The `session_connect_attrs` table contains connection attributes for all sessions. To see connection attributes only for the current session, and other sessions associated with the session account, use the `session_account_connect_attrs` table.

The `session_connect_attrs` table has these columns:

- `PROCESSLIST_ID`

  The connection identifier for the session.

- `ATTR_NAME`

  The attribute name.

- `ATTR_VALUE`

  The attribute value.

- `ORDINAL_POSITION`

  The order in which the attribute was added to the set of connection attributes.

The `session_connect_attrs` table has these indexes:

- Primary key on (`PROCESSLIST_ID`, `ATTR_NAME`)

`TRUNCATE TABLE` is not permitted for the `session_connect_attrs` table.

10.10 Performance Schema User-Defined Variable Tables

The Performance Schema provides a `user_variables_by_thread` table that exposes user-defined variables. These are variables defined within a specific session and include a `@` character preceding the name; see User-Defined Variables.

The `user_variables_by_thread` table has these columns:

- `THREAD_ID`

  The thread identifier of the session in which the variable is defined.

- `VARIABLE_NAME`

  The variable name, without the leading `@` character.

- `VARIABLE_VALUE`

  The variable value.

The `user_variables_by_thread` table has these indexes:

- Primary key on (`THREAD_ID`, `VARIABLE_NAME`)

`TRUNCATE TABLE` is not permitted for the `user_variables_by_thread` table.

10.11 Performance Schema Replication Tables

The Performance Schema provides tables that expose replication information. This is similar to the information available from the `SHOW REPLICA STATUS` statement, but representation in table form is more accessible and has usability benefits:

- `SHOW REPLICA STATUS` output is useful for visual inspection, but not so much for programmatic use. By contrast, using the Performance Schema tables, information about replica status can be searched using general `SELECT` queries, including complex `WHERE` conditions, joins, and so forth.
- Query results can be saved in tables for further analysis, or assigned to variables and thus used in stored procedures.
- The replication tables provide better diagnostic information. For multithreaded replica operation, `SHOW REPLICA STATUS` reports all coordinator and worker thread errors using the `Last_SQL_Errno` and `Last_SQL_Error` fields, so only the most recent of those errors is visible and information can be lost. The replication tables store errors on a per-thread basis without loss of information.
- The last seen transaction is visible in the replication tables on a per-worker basis. This is information not available from `SHOW REPLICA STATUS`.
- Developers familiar with the Performance Schema interface can extend the replication tables to provide additional information by adding rows to the tables.

Replication Table Descriptions

The Performance Schema provides the following replication-related tables:

- Tables that contain information about the connection of the replica to the source:
  - `replication_connection_configuration`: Configuration parameters for connecting to the source
  - `replication_connection_status`: Current status of the connection to the source
  - `replication_asynchronous_connection_failover`: Source lists for the asynchronous connection failover mechanism
- Tables that contain general (not thread-specific) information about the transaction applier:
  - `replication_applier_configuration`: Configuration parameters for the transaction applier on the replica.
  - `replication_applier_status`: Current status of the transaction applier on the replica.
- Tables that contain information about specific threads responsible for applying transactions received from the source:
  - `replication_applier_status_by_coordinator`: Status of the coordinator thread (empty unless the replica is multithreaded).
  - `replication_applier_status_by_worker`: Status of the applier thread or worker threads if the replica is multithreaded.
- Tables that contain information about channel based replication filters:

  - `replication_applier_filters`: Provides information about the replication filters configured on specific replication channels.
  - `replication_applier_global_filters`: Provides information about global replication filters, which apply to all replication channels.
- Tables that contain information about Group Replication members:
  - `replication_group_members`: Provides network and status information for group members.
  - `replication_group_member_stats`: Provides statistical information about group members and transactions in which they participate.

  For more information see Monitoring Group Replication.

The following Performance Schema replication tables continue to be populated when the Performance Schema is disabled:

- `replication_connection_configuration`
- `replication_connection_status`
- `replication_asynchronous_connection_failover`
- `replication_applier_configuration`
- `replication_applier_status`
- `replication_applier_status_by_coordinator`
- `replication_applier_status_by_worker`

The exception is local timing information (start and end timestamps for transactions) in the replication tables `replication_connection_status`, `replication_applier_status_by_coordinator`, and `replication_applier_status_by_worker`. This information is not collected when the Performance Schema is disabled.

The following sections describe each replication table in more detail, including the correspondence between the columns produced by `SHOW REPLICA STATUS` and the replication table columns in which the same information appears.

The remainder of this introduction to the replication tables describes how the Performance Schema populates them and which fields from `SHOW REPLICA STATUS` are not represented in the tables.

Replication Table Life Cycle

The Performance Schema populates the replication tables as follows:

- Prior to execution of `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO`, the tables are empty.
- After `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO`, the configuration parameters can be seen in the tables. At this time, there are no active replication threads, so the `THREAD_ID` columns are `NULL` and the `SERVICE_STATE` columns have a value of `OFF`.
- After `START REPLICA` (or before MySQL 8.0.22, `START SLAVE`), non-`NULL` `THREAD_ID` values can be seen. Threads that are idle or active have a `SERVICE_STATE` value of `ON`. The thread that connects to the source has a value of `CONNECTING` while it establishes the connection, and `ON` thereafter as long as the connection lasts.

- After `STOP REPLICA`, the `THREAD_ID` columns become `NULL` and the `SERVICE_STATE` columns for threads that no longer exist have a value of `OFF`.
- The tables are preserved after `STOP REPLICA` or threads stopping due to an error.
- The `replication_applier_status_by_worker` table is nonempty only when the replica is operating in multithreaded mode. That is, if the `replica_parallel_workers` or `slave_parallel_workers` system variable is greater than 0, this table is populated when `START REPLICA` is executed, and the number of rows shows the number of workers.

Replica Status Information Not In the Replication Tables

The information in the Performance Schema replication tables differs somewhat from the information available from `SHOW REPLICA STATUS` because the tables are oriented toward use of global transaction identifiers (GTIDs), not file names and positions, and they represent server UUID values, not server ID values. Due to these differences, several `SHOW REPLICA STATUS` columns are not preserved in the Performance Schema replication tables, or are represented a different way:

- The following fields refer to file names and positions and are not preserved:

```
Master_Log_File
Read_Master_Log_Pos
Relay_Log_File
Relay_Log_Pos
Relay_Master_Log_File
Exec_Master_Log_Pos
Until_Condition
Until_Log_File
Until_Log_Pos
```

- The `Master_Info_File` field is not preserved. It refers to the `master.info` file used for the replica's source metadata repository, which has been superseded by the use of crash-safe tables for the repository.
- The following fields are based on `server_id`, not `server_uuid`, and are not preserved:

```
Master_Server_Id
Replicate_Ignore_Server_Ids
```

- The `Skip_Counter` field is based on event counts, not GTIDs, and is not preserved.
- These error fields are aliases for `Last_SQL_Errno` and `Last_SQL_Error`, so they are not preserved:

```
Last_Errno
Last_Error
```

  In the Performance Schema, this error information is available in the `LAST_ERROR_NUMBER` and `LAST_ERROR_MESSAGE` columns of the `replication_applier_status_by_worker` table (and `replication_applier_status_by_coordinator` if the replica is multithreaded). Those tables provide more specific per-thread error information than is available from `Last_Errno` and `Last_Error`.

- Fields that provide information about command-line filtering options is not preserved:

```
Replicate_Do_DB
Replicate_Ignore_DB
Replicate_Do_Table
Replicate_Ignore_Table
Replicate_Wild_Do_Table
Replicate_Wild_Ignore_Table
```

- The `Replica_IO_State` and `Replica_SQL_Running_State` fields are not preserved. If needed, these values can be obtained from the process list by using the `THREAD_ID` column of the appropriate replication table and joining it with the `ID` column in the `INFORMATION_SCHEMA PROCESSLIST` table to select the `STATE` column of the latter table.
- The `Executed_Gtid_Set` field can show a large set with a great deal of text. Instead, the Performance Schema tables show GTIDs of transactions that are currently being applied by the replica. Alternatively, the set of executed GTIDs can be obtained from the value of the `gtid_executed` system variable.
- The `Seconds_Behind_Master` and `Relay_Log_Space` fields are in to-be-decided status and are not preserved.

Replication Channels

The first column of the replication Performance Schema tables is `CHANNEL_NAME`. This enables the tables to be viewed per replication channel. In a non-multisource replication setup there is a single default replication channel. When you are using multiple replication channels on a replica, you can filter the tables per replication channel to monitor a specific replication channel. See Replication Channels and Monitoring Multi-Source Replication for more information.

10.11.1 The binary_log_transaction_compression_stats Table

This table shows statistical information for transaction payloads written to the binary log and relay log, and can be used to calculate the effects of enabling binary log transaction compression. For information on binary log transaction compression, see Binary Log Transaction Compression.

The `binary_log_transaction_compression_stats` table is populated only when the server instance has a binary log, and the system variable `binlog_transaction_compression` is set to `ON`. The statistics cover all transactions written to the binary log and relay log from the time the server was started or the table was truncated. Compressed transactions are grouped by the compression algorithm used, and uncompressed transactions are grouped together with the compression algorithm stated as `NONE`, so the compression ratio can be calculated.

The `binary_log_transaction_compression_stats` table has these columns:

- `LOG_TYPE`

  Whether these transactions were written to the binary log or relay log.

- `COMPRESSION_TYPE`

  The compression algorithm used to compress the transaction payloads. `NONE` means the payloads for these transactions were not compressed, which is correct in a number of situations (see Binary Log Transaction Compression).

- `TRANSACTION_COUNTER`

  The number of transactions written to this log type with this compression type.

- `COMPRESSED_BYTES`

  The total number of bytes that were compressed and then written to this log type with this compression type, counted after compression.

- `UNCOMPRESSED_BYTES`

  The total number of bytes before compression for this log type and this compression type.

- `COMPRESSION_PERCENTAGE`

  The compression ratio for this log type and this compression type, expressed as a percentage.

- `FIRST_TRANSACTION_ID`

  The ID of the first transaction that was written to this log type with this compression type.

- `FIRST_TRANSACTION_COMPRESSED_BYTES`

  The total number of bytes that were compressed and then written to the log for the first transaction, counted after compression.

- `FIRST_TRANSACTION_UNCOMPRESSED_BYTES`

  The total number of bytes before compression for the first transaction.

- `FIRST_TRANSACTION_TIMESTAMP`

  The timestamp when the first transaction was written to the log.

- `LAST_TRANSACTION_ID`

  The ID of the most recent transaction that was written to this log type with this compression type.

- `LAST_TRANSACTION_COMPRESSED_BYTES`

  The total number of bytes that were compressed and then written to the log for the most recent transaction, counted after compression.

- `LAST_TRANSACTION_UNCOMPRESSED_BYTES`

  The total number of bytes before compression for the most recent transaction.

- `LAST_TRANSACTION_TIMESTAMP`

  The timestamp when the most recent transaction was written to the log.

The `binary_log_transaction_compression_stats` table has no indexes.

`TRUNCATE TABLE` is permitted for the `binary_log_transaction_compression_stats` table.

10.11.2 The replication_applier_configuration Table

This table shows the configuration parameters that affect transactions applied by the replica. Parameters stored in the table can be changed at runtime with the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23).

The `replication_applier_configuration` table has these columns:

- `CHANNEL_NAME`

  The replication channel which this row is displaying. There is always a default replication channel, and more replication channels can be added. See Replication Channels for more information.

- `DESIRED_DELAY`

  The number of seconds that the replica must lag the source. (`CHANGE REPLICATION SOURCE TO` option: `SOURCE_DELAY`, `CHANGE MASTER TO` option: `MASTER_DELAY`) See Delayed Replication for more information.

- PRIVILEGE_CHECKS_USER

  The user account that provides the security context for the channel (CHANGE REPLICATION SOURCE TO option: PRIVILEGE_CHECKS_USER, CHANGE MASTER TO option: PRIVILEGE_CHECKS_USER). This is escaped so that it can be copied into an SQL statement to execute individual transactions. See Replication Privilege Checks for more information.

- REQUIRE_ROW_FORMAT

  Whether the channel accepts only row-based events (CHANGE REPLICATION SOURCE TO option: REQUIRE_ROW_FORMAT, CHANGE MASTER TO option: REQUIRE_ROW_FORMAT). See Replication Privilege Checks for more information.

- REQUIRE_TABLE_PRIMARY_KEY_CHECK

  Whether the channel requires primary keys always, never, or according to the source's setting (CHANGE REPLICATION SOURCE TO option: REQUIRE_TABLE_PRIMARY_KEY_CHECK, CHANGE MASTER TO option: REQUIRE_TABLE_PRIMARY_KEY_CHECK). See Replication Privilege Checks for more information.

- ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS_TYPE

  Whether the channel assigns a GTID to replicated transactions that do not already have one (CHANGE REPLICATION SOURCE TO option: ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS, CHANGE MASTER TO option: ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS). OFF means no GTIDs are assigned. LOCAL means a GTID is assigned that includes the replica's own UUID (the server_uuid setting). UUID means a GTID is assigned that includes a manually set UUID. See Replication From a Source Without GTIDs to a Replica With GTIDs for more information.

- ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS_VALUE

  The UUID that is used as part of the GTIDs assigned to anonymous transactions (CHANGE REPLICATION SOURCE TO option: ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS, CHANGE MASTER TO option: ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS). See Replication From a Source Without GTIDs to a Replica With GTIDs for more information.

The replication_applier_configuration table has these indexes:

- Primary key on (CHANNEL_NAME)

TRUNCATE TABLE is not permitted for the replication_applier_configuration table.

The following table shows the correspondence between replication_applier_configuration columns and SHOW REPLICA STATUS columns.

replication_applier_configuration Column	SHOW REPLICA STATUS Column
DESIRED_DELAY	SQL_Delay

10.11.3 The replication_applier_status Table

This table shows the current general transaction execution status on the replica. The table provides information about general aspects of transaction applier status that are not specific to any thread involved. Thread-specific status information is available in the replication_applier_status_by_coordinator table (and replication_applier_status_by_worker if the replica is multithreaded).

The `replication_applier_status` table has these columns:

- `CHANNEL_NAME`

  The replication channel which this row is displaying. There is always a default replication channel, and more replication channels can be added. See Replication Channels for more information.

- `SERVICE_STATE`

  Shows `ON` when the replication channel's applier threads are active or idle, `OFF` means that the applier threads are not active.

- `REMAINING_DELAY`

  If the replica is waiting for `DESIRED_DELAY` seconds to pass since the source applied a transaction, this field contains the number of delay seconds remaining. At other times, this field is `NULL`. (The `DESIRED_DELAY` value is stored in the `replication_applier_configuration` table.) See Delayed Replication for more information.

- `COUNT_TRANSACTIONS_RETRIES`

  Shows the number of retries that were made because the replication SQL thread failed to apply a transaction. The maximum number of retries for a given transaction is set by the system variable `replica_transaction_retries` and `slave_transaction_retries`. The `replication_applier_status_by_worker` table shows detailed information on transaction retries for a single-threaded or multithreaded replica.

The `replication_applier_status` table has these indexes:

- Primary key on (`CHANNEL_NAME`)

`TRUNCATE TABLE` is not permitted for the `replication_applier_status` table.

The following table shows the correspondence between `replication_applier_status` columns and `SHOW REPLICA STATUS` columns.

`replication_applier_status` Column	**`SHOW REPLICA STATUS` Column**
`SERVICE_STATE`	None
`REMAINING_DELAY`	`SQL_Remaining_Delay`

10.11.4 The replication_applier_status_by_coordinator Table

For a multithreaded replica, the replica uses multiple worker threads and a coordinator thread to manage them, and this table shows the status of the coordinator thread. For a single-threaded replica, this table is empty. For a multithreaded replica, the `replication_applier_status_by_worker` table shows the status of the worker threads. This table provides information about the last transaction which was buffered by the coordinator thread to a worker's queue, as well as the transaction it is currently buffering. The start timestamp refers to when this thread read the first event of the transaction from the relay log to buffer it to a worker's queue, while the end timestamp refers to when the last event finished buffering to the worker's queue.

The `replication_applier_status_by_coordinator` table has these columns:

- `CHANNEL_NAME`

  The replication channel which this row is displaying. There is always a default replication channel, and more replication channels can be added. See Replication Channels for more information.

- `THREAD_ID`

  The SQL/coordinator thread ID.

- `SERVICE_STATE`

  `ON` (thread exists and is active or idle) or `OFF` (thread no longer exists).

- `LAST_ERROR_NUMBER`, `LAST_ERROR_MESSAGE`

  The error number and error message of the most recent error that caused the SQL/coordinator thread to stop. An error number of 0 and message which is an empty string means “no error”. If the `LAST_ERROR_MESSAGE` value is not empty, the error values also appear in the replica's error log.

  Issuing `RESET MASTER` or `RESET REPLICA` resets the values shown in these columns.

  All error codes and messages displayed in the `LAST_ERROR_NUMBER` and `LAST_ERROR_MESSAGE` columns correspond to error values listed in Server Error Message Reference.

- `LAST_ERROR_TIMESTAMP`

  A timestamp in `'YYYY-MM-DD hh:mm:ss[.fraction]'` format that shows when the most recent SQL/coordinator error occurred.

- `LAST_PROCESSED_TRANSACTION`

  The global transaction ID (GTID) of the last transaction processed by this coordinator.

- `LAST_PROCESSED_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP`

  A timestamp in `'YYYY-MM-DD hh:mm:ss[.fraction]'` format that shows when the last transaction processed by this coordinator was committed on the original source.

- `LAST_PROCESSED_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP`

  A timestamp in `'YYYY-MM-DD hh:mm:ss[.fraction]'` format that shows when the last transaction processed by this coordinator was committed on the immediate source.

- `LAST_PROCESSED_TRANSACTION_START_BUFFER_TIMESTAMP`

  A timestamp in `'YYYY-MM-DD hh:mm:ss[.fraction]'` format that shows when this coordinator thread started writing the last transaction to the buffer of a worker thread.

- `LAST_PROCESSED_TRANSACTION_END_BUFFER_TIMESTAMP`

  A timestamp in `'YYYY-MM-DD hh:mm:ss[.fraction]'` format that shows when the last transaction was written to the buffer of a worker thread by this coordinator thread.

- `PROCESSING_TRANSACTION`

  The global transaction ID (GTID) of the transaction that this coordinator thread is currently processing.

- `PROCESSING_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP`

  A timestamp in `'YYYY-MM-DD hh:mm:ss[.fraction]'` format that shows when the currently processing transaction was committed on the original source.

- `PROCESSING_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP`

A timestamp in *'YYYY-MM-DD hh:mm:ss[.fraction]'* format that shows when the currently processing transaction was committed on the immediate source.

- PROCESSING_TRANSACTION_START_BUFFER_TIMESTAMP

  A timestamp in *'YYYY-MM-DD hh:mm:ss[.fraction]'* format that shows when this coordinator thread started writing the currently processing transaction to the buffer of a worker thread.

When the Performance Schema is disabled, local timing information is not collected, so the fields showing the start and end timestamps for buffered transactions are zero.

The replication_applier_status_by_coordinator table has these indexes:

- Primary key on (CHANNEL_NAME)
- Index on (THREAD_ID)

The following table shows the correspondence between replication_applier_status_by_coordinator columns and SHOW REPLICA STATUS columns.

replication_applier_status_by_coordinator Column	SHOW REPLICA STATUS Column
THREAD_ID	None
SERVICE_STATE	Replica_SQL_Running
LAST_ERROR_NUMBER	Last_SQL_Errno
LAST_ERROR_MESSAGE	Last_SQL_Error
LAST_ERROR_TIMESTAMP	Last_SQL_Error_Timestamp

10.11.5 The replication_applier_status_by_worker Table

This table provides details of the transactions handled by applier threads on a replica or Group Replication group member. For a single-threaded replica, data is shown for the replica's single applier thread. For a multithreaded replica, data is shown individually for each applier thread. The applier threads on a multithreaded replica are sometimes called workers. The number of applier threads on a replica or Group Replication group member is set by the replica_parallel_workers or slave_parallel_workers system variable, which is set to zero for a single-threaded replica. A multithreaded replica also has a coordinator thread to manage the applier threads, and the status of this thread is shown in the replication_applier_status_by_coordinator table.

All error codes and messages displayed in the columns relating to errors correspond to error values listed in Server Error Message Reference.

When the Performance Schema is disabled, local timing information is not collected, so the fields showing the start and end timestamps for applied transactions are zero. The start timestamps in this table refer to when the worker started applying the first event, and the end timestamps refer to when the last event of the transaction was applied.

When a replica is restarted by a START REPLICA statement, the columns beginning APPLYING_TRANSACTION are reset. Before MySQL 8.0.13, these columns were not reset on a replica that was operating in single-threaded mode, only on a multithreaded replica.

The replication_applier_status_by_worker table has these columns:

- `CHANNEL_NAME`

  The replication channel which this row is displaying. There is always a default replication channel, and more replication channels can be added. See Replication Channels for more information.

- `WORKER_ID`

  The worker identifier (same value as the `id` column in the `mysql.slave_worker_info` table). After `STOP REPLICA`, the `THREAD_ID` column becomes `NULL`, but the `WORKER_ID` value is preserved.

- `THREAD_ID`

  The worker thread ID.

- `SERVICE_STATE`

  `ON` (thread exists and is active or idle) or `OFF` (thread no longer exists).

- `LAST_ERROR_NUMBER`, `LAST_ERROR_MESSAGE`

  The error number and error message of the most recent error that caused the worker thread to stop. An error number of 0 and message of the empty string mean “no error”. If the `LAST_ERROR_MESSAGE` value is not empty, the error values also appear in the replica's error log.

  Issuing `RESET MASTER` or `RESET REPLICA` resets the values shown in these columns.

- `LAST_ERROR_TIMESTAMP`

  A timestamp in `'YYYY-MM-DD hh:mm:ss[.fraction]'` format that shows when the most recent worker error occurred.

- `LAST_APPLIED_TRANSACTION`

  The global transaction ID (GTID) of the last transaction applied by this worker.

- `LAST_APPLIED_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP`

  A timestamp in `'YYYY-MM-DD hh:mm:ss[.fraction]'` format that shows when the last transaction applied by this worker was committed on the original source.

- `LAST_APPLIED_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP`

  A timestamp in `'YYYY-MM-DD hh:mm:ss[.fraction]'` format that shows when the last transaction applied by this worker was committed on the immediate source.

- `LAST_APPLIED_TRANSACTION_START_APPLY_TIMESTAMP`

  A timestamp in `'YYYY-MM-DD hh:mm:ss[.fraction]'` format that shows when this worker started applying the last applied transaction.

- `LAST_APPLIED_TRANSACTION_END_APPLY_TIMESTAMP`

  A timestamp in `'YYYY-MM-DD hh:mm:ss[.fraction]'` format that shows when this worker finished applying the last applied transaction.

- `APPLYING_TRANSACTION`

  The global transaction ID (GTID) of the transaction this worker is currently applying.

- APPLYING_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP

  A timestamp in '*YYYY-MM-DD hh:mm:ss*[.*fraction*]' format that shows when the transaction this worker is currently applying was committed on the original source.

- APPLYING_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP

  A timestamp in '*YYYY-MM-DD hh:mm:ss*[.*fraction*]' format that shows when the transaction this worker is currently applying was committed on the immediate source.

- APPLYING_TRANSACTION_START_APPLY_TIMESTAMP

  A timestamp in '*YYYY-MM-DD hh:mm:ss*[.*fraction*]' format that shows when this worker started its first attempt to apply the transaction that is currently being applied. Before MySQL 8.0.13, this timestamp was refreshed when a transaction was retried due to a transient error, so it showed the timestamp for the most recent attempt to apply the transaction.

- LAST_APPLIED_TRANSACTION_RETRIES_COUNT

  The number of times the last applied transaction was retried by the worker after the first attempt. If the transaction was applied at the first attempt, this number is zero.

- LAST_APPLIED_TRANSACTION_LAST_TRANSIENT_ERROR_NUMBER

  The error number of the last transient error that caused the transaction to be retried.

- LAST_APPLIED_TRANSACTION_LAST_TRANSIENT_ERROR_MESSAGE

  The message text for the last transient error that caused the transaction to be retried.

- LAST_APPLIED_TRANSACTION_LAST_TRANSIENT_ERROR_TIMESTAMP

  A timestamp in '*YYYY-MM-DD hh:mm:ss*[.*fraction*]' format for the last transient error that caused the transaction to be retried.

- APPLYING_TRANSACTION_RETRIES_COUNT

  The number of times the transaction that is currently being applied was retried until this moment. If the transaction was applied at the first attempt, this number is zero.

- APPLYING_TRANSACTION_LAST_TRANSIENT_ERROR_NUMBER

  The error number of the last transient error that caused the current transaction to be retried.

- APPLYING_TRANSACTION_LAST_TRANSIENT_ERROR_MESSAGE

  The message text for the last transient error that caused the current transaction to be retried.

- APPLYING_TRANSACTION_LAST_TRANSIENT_ERROR_TIMESTAMP

  A timestamp in '*YYYY-MM-DD hh:mm:ss*[.*fraction*]' format for the last transient error that caused the current transaction to be retried.

The replication_applier_status_by_worker table has these indexes:

- Primary key on (CHANNEL_NAME, WORKER_ID)
- Index on (THREAD_ID)

The following table shows the correspondence between `replication_applier_status_by_worker` columns and `SHOW REPLICA STATUS` columns.

`replication_applier_status_by_worker` Column	`SHOW REPLICA STATUS` Column
`WORKER_ID`	None
`THREAD_ID`	None
`SERVICE_STATE`	None
`LAST_ERROR_NUMBER`	`Last_SQL_Errno`
`LAST_ERROR_MESSAGE`	`Last_SQL_Error`
`LAST_ERROR_TIMESTAMP`	`Last_SQL_Error_Timestamp`

10.11.6 The replication_applier_filters Table

This table shows the replication channel specific filters configured on this replica. Each row provides information on a replication channel's configured type of filter. The `replication_applier_filters` table has these columns:

- `CHANNEL_NAME`

  The name of replication channel with a replication filter configured.

- `FILTER_NAME`

  The type of replication filter that has been configured for this replication channel.

- `FILTER_RULE`

  The rules configured for the replication filter type using either `--replicate-*` command options or `CHANGE REPLICATION FILTER`.

- `CONFIGURED_BY`

  The method used to configure the replication filter, can be one of:

  - `CHANGE_REPLICATION_FILTER` configured by a global replication filter using a `CHANGE REPLICATION FILTER` statement.
  - `STARTUP_OPTIONS` configured by a global replication filter using a `--replicate-*` option.
  - `CHANGE_REPLICATION_FILTER_FOR_CHANNEL` configured by a channel specific replication filter using a `CHANGE REPLICATION FILTER FOR CHANNEL` statement.
  - `STARTUP_OPTIONS_FOR_CHANNEL` configured by a channel specific replication filter using a `--replicate-*` option.

- `ACTIVE_SINCE`

  Timestamp of when the replication filter was configured.

- `COUNTER`

  The number of times the replication filter has been used since it was configured.

10.11.7 The replication_applier_global_filters Table

This table shows the global replication filters configured on this replica. The `replication_applier_global_filters` table has these columns:

- `FILTER_NAME`

  The type of replication filter that has been configured.

- `FILTER_RULE`

  The rules configured for the replication filter type using either `--replicate-*` command options or `CHANGE REPLICATION FILTER`.

- `CONFIGURED_BY`

  The method used to configure the replication filter, can be one of:

  - `CHANGE_REPLICATION_FILTER` configured by a global replication filter using a `CHANGE REPLICATION FILTER` statement.
  - `STARTUP_OPTIONS` configured by a global replication filter using a `--replicate-*` option.

- `ACTIVE_SINCE`

  Timestamp of when the replication filter was configured.

10.11.8 The replication_asynchronous_connection_failover Table

This table holds the replica's source lists for each replication channel for the asynchronous connection failover mechanism. The asynchronous connection failover mechanism automatically establishes an asynchronous (source to replica) replication connection to a new source from the appropriate list after the existing connection from the replica to its source fails. When asynchronous connection failover is enabled for a group of replicas managed by Group Replication, the source lists are broadcast to all group members when they join, and also when the lists change.

You set and manage source lists using the `asynchronous_connection_failover_add_source` and `asynchronous_connection_failover_delete_source` functions to add and remove replication source servers from the source list for a replication channel. To add and remove managed groups of servers, use the `asynchronous_connection_failover_add_managed` and `asynchronous_connection_failover_delete_managed` functions instead.

For more information, see Switching Sources and Replicas with Asynchronous Connection Failover.

The `replication_asynchronous_connection_failover` table has these columns:

- `CHANNEL_NAME`

  The replication channel for which this replication source server is part of the source list. If this channel's connection to its current source fails, this replication source server is one of its potential new sources.

- `HOST`

  The host name for this replication source server.

- `PORT`

  The port number for this replication source server.

- `NETWORK_NAMESPACE`

The network namespace for this replication source server. If this value is empty, connections use the default (global) namespace.

- `WEIGHT`

  The priority of this replication source server in the replication channel's source list. The weight is from 1 to 100, with 100 being the highest, and 50 being the default. When the asynchronous connection failover mechanism activates, the source with the highest weight setting among the alternative sources listed in the source list for the channel is chosen for the first connection attempt. If this attempt does not work, the replica tries with all the listed sources in descending order of weight, then starts again from the highest weighted source. If multiple sources have the same weight, the replica orders them randomly.

- `MANAGED_NAME`

  The identifier for the managed group that the server is a part of. For the `GroupReplication` managed service, the identifier is the value of the `group_replication_group_name` system variable.

The `replication_asynchronous_connection_failover` table has these indexes:

- Primary key on (`CHANNEL_NAME, HOST, PORT, NETWORK_NAMESPACE, MANAGED_NAME`)

`TRUNCATE TABLE` is not permitted for the `replication_asynchronous_connection_failover` table.

10.11.9 The replication_asynchronous_connection_failover_managed Table

This table holds configuration information used by the replica's asynchronous connection failover mechanism to handle managed groups, including Group Replication topologies.

When you add a group member to the source list and define it as part of a managed group, the asynchronous connection failover mechanism updates the source list to keep it in line with membership changes, adding and removing group members automatically as they join or leave. When asynchronous connection failover is enabled for a group of replicas managed by Group Replication, the source lists are broadcast to all group members when they join, and also when the lists change.

The asynchronous connection failover mechanism fails over the connection if another available server on the source list has a higher priority (weight) setting. For a managed group, a source's weight is assigned depending on whether it is a primary or a secondary server. So assuming that you set up the managed group to give a higher weight to a primary and a lower weight to a secondary, when the primary changes, the higher weight is assigned to the new primary, so the replica changes over the connection to it. The asynchronous connection failover mechanism additionally changes connection if the currently connected managed source server leaves the managed group, or is no longer in the majority in the managed group. For more information, see Switching Sources and Replicas with Asynchronous Connection Failover.

The `replication_asynchronous_connection_failover_managed` table has these columns:

- `CHANNEL_NAME`

  The replication channel where the servers for this managed group operate.

- `MANAGED_NAME`

  The identifier for the managed group. For the `GroupReplication` managed service, the identifier is the value of the `group_replication_group_name` system variable.

- `MANAGED_TYPE`

The type of managed service that the asynchronous connection failover mechanism provides for this group. The only value currently available is `GroupReplication`.

- `CONFIGURATION`

  The configuration information for this managed group. For the `GroupReplication` managed service, the configuration shows the weights assigned to the group's primary server and to the group's secondary servers. For example: `{"Primary_weight": 80, "Secondary_weight": 60}`

  - `Primary_weight`: Integer between 0 and 100. Default value is 80.
  - `Secondary_weight`: Integer between 0 and 100. Default value is 60.

The `replication_asynchronous_connection_failover_managed` table has these indexes:

- Primary key on (`CHANNEL_NAME, MANAGED_NAME`)

`TRUNCATE TABLE` is not permitted for the `replication_asynchronous_connection_failover_managed` table.

10.11.10 The replication_connection_configuration Table

This table shows the configuration parameters used by the replica for connecting to the source. Parameters stored in the table can be changed at runtime with the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23).

Compared to the `replication_connection_status` table, `replication_connection_configuration` changes less frequently. It contains values that define how the replica connects to the source and that remain constant during the connection, whereas `replication_connection_status` contains values that change during the connection.

The `replication_connection_configuration` table has the following columns. The column descriptions indicate the corresponding `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` options from which the column values are taken, and the table given later in this section shows the correspondence between `replication_connection_configuration` columns and `SHOW REPLICA STATUS` columns.

- `CHANNEL_NAME`

  The replication channel which this row is displaying. There is always a default replication channel, and more replication channels can be added. See Replication Channels for more information. (`CHANGE REPLICATION SOURCE TO` option: `FOR CHANNEL`, `CHANGE MASTER TO` option: `FOR CHANNEL`)

- `HOST`

  The host name of the source that the replica is connected to. (`CHANGE REPLICATION SOURCE TO` option: `SOURCE_HOST`, `CHANGE MASTER TO` option: `MASTER_HOST`)

- `PORT`

  The port used to connect to the source. (`CHANGE REPLICATION SOURCE TO` option: `SOURCE_PORT`, `CHANGE MASTER TO` option: `MASTER_PORT`)

- `USER`

  The user name of the replication user account used to connect to the source. (`CHANGE REPLICATION SOURCE TO` option: `SOURCE_USER`, `CHANGE MASTER TO` option: `MASTER_USER`)

- NETWORK_INTERFACE

  The network interface that the replica is bound to, if any. (CHANGE REPLICATION SOURCE TO option: SOURCE_BIND, CHANGE MASTER TO option: MASTER_BIND)

- AUTO_POSITION

  1 if GTID auto-positioning is in use; otherwise 0. (CHANGE REPLICATION SOURCE TO option: SOURCE_AUTO_POSITION, CHANGE MASTER TO option: MASTER_AUTO_POSITION)

- SSL_ALLOWED, SSL_CA_FILE, SSL_CA_PATH, SSL_CERTIFICATE, SSL_CIPHER, SSL_KEY, SSL_VERIFY_SERVER_CERTIFICATE, SSL_CRL_FILE, SSL_CRL_PATH

  These columns show the SSL parameters used by the replica to connect to the source, if any.

  SSL_ALLOWED has these values:

  - Yes if an SSL connection to the source is permitted
  - No if an SSL connection to the source is not permitted
  - Ignored if an SSL connection is permitted but the replica does not have SSL support enabled

  (CHANGE REPLICATION SOURCE TO options for the other SSL columns: SOURCE_SSL_CA, SOURCE_SSL_CAPATH, SOURCE_SSL_CERT, SOURCE_SSL_CIPHER, SOURCE_SSL_CRL, SOURCE_SSL_CRLPATH, SOURCE_SSL_KEY, SOURCE_SSL_VERIFY_SERVER_CERT.

  CHANGE MASTER TO options for the other SSL columns: MASTER_SSL_CA, MASTER_SSL_CAPATH, MASTER_SSL_CERT, MASTER_SSL_CIPHER, MASTER_SSL_CRL, MASTER_SSL_CRLPATH, MASTER_SSL_KEY, MASTER_SSL_VERIFY_SERVER_CERT.

- CONNECTION_RETRY_INTERVAL

  The number of seconds between connect retries. (CHANGE REPLICATION SOURCE TO option: SOURCE_CONNECT_RETRY, CHANGE MASTER TO option: MASTER_CONNECT_RETRY)

- CONNECTION_RETRY_COUNT

  The number of times the replica can attempt to reconnect to the source in the event of a lost connection. (CHANGE REPLICATION SOURCE TO option: SOURCE_RETRY_COUNT, CHANGE MASTER TO option: MASTER_RETRY_COUNT)

- HEARTBEAT_INTERVAL

  The replication heartbeat interval on a replica, measured in seconds. (CHANGE REPLICATION SOURCE TO option: SOURCE_HEARTBEAT_PERIOD, CHANGE MASTER TO option: MASTER_HEARTBEAT_PERIOD)

- TLS_VERSION

  The list of TLS protocol versions that are permitted by the replica for the replication connection. For TLS version information, see Encrypted Connection TLS Protocols and Ciphers. (CHANGE REPLICATION SOURCE TO option: SOURCE_TLS_VERSION, CHANGE MASTER TO option: MASTER_TLS_VERSION)

- TLS_CIPHERSUITES

  The list of ciphersuites that are permitted by the replica for the replication connection. For TLS ciphersuite information, see Encrypted Connection TLS Protocols and Ciphers. (CHANGE

REPLICATION SOURCE TO option: SOURCE_TLS_CIPHERSUITES, CHANGE MASTER TO option: MASTER_TLS_CIPHERSUITES)

- PUBLIC_KEY_PATH

  The path name to a file containing a replica-side copy of the public key required by the source for RSA key pair-based password exchange. The file must be in PEM format. This column applies to replicas that authenticate with the sha256_password or caching_sha2_password authentication plugin. (CHANGE REPLICATION SOURCE TO option: SOURCE_PUBLIC_KEY_PATH, CHANGE MASTER TO option: MASTER_PUBLIC_KEY_PATH)

  If PUBLIC_KEY_PATH is given and specifies a valid public key file, it takes precedence over GET_PUBLIC_KEY.

- GET_PUBLIC_KEY

  Whether to request from the source the public key required for RSA key pair-based password exchange. This column applies to replicas that authenticate with the caching_sha2_password authentication plugin. For that plugin, the source does not send the public key unless requested. (CHANGE REPLICATION SOURCE TO option: GET_SOURCE_PUBLIC_KEY, CHANGE MASTER TO option: GET_MASTER_PUBLIC_KEY)

  If PUBLIC_KEY_PATH is given and specifies a valid public key file, it takes precedence over GET_PUBLIC_KEY.

- NETWORK_NAMESPACE

  The network namespace name; empty if the connection uses the default (global) namespace. For information about network namespaces, see Network Namespace Support. This column was added in MySQL 8.0.22.

- COMPRESSION_ALGORITHM

  The permitted compression algorithms for connections to the source. (CHANGE REPLICATION SOURCE TO option: SOURCE_COMPRESSION_ALGORITHMS, CHANGE MASTER TO option: MASTER_COMPRESSION_ALGORITHMS)

  For more information, see Connection Compression Control.

  This column was added in MySQL 8.0.18.

- ZSTD_COMPRESSION_LEVEL

  The compression level to use for connections to the source that use the zstd compression algorithm. (CHANGE REPLICATION SOURCE TO option: SOURCE_ZSTD_COMPRESSION_LEVEL, CHANGE MASTER TO option: MASTER_ZSTD_COMPRESSION_LEVEL)

  For more information, see Connection Compression Control.

  This column was added in MySQL 8.0.18.

- SOURCE_CONNECTION_AUTO_FAILOVER

  Whether the asynchronous connection failover mechanism is activated for this replication channel. (CHANGE REPLICATION SOURCE TO option: SOURCE_CONNECTION_AUTO_FAILOVER, CHANGE MASTER TO option: SOURCE_CONNECTION_AUTO_FAILOVER)

  For more information, see Switching Sources and Replicas with Asynchronous Connection Failover.

This column was added in MySQL 8.0.22.

- `GTID_ONLY`

  Indicates if this channel only uses GTIDs for the transaction queueing and application process and for recovery, and does not persist binary log and relay log file names and file positions in the replication metadata repositories. (`CHANGE REPLICATION SOURCE TO` option: `GTID_ONLY`, `CHANGE MASTER TO` option: `GTID_ONLY`)

  For more information, see GTIDs and Group Replication.

  This column was added in MySQL 8.0.27.

The `replication_connection_configuration` table has these indexes:

- Primary key on (`CHANNEL_NAME`)

`TRUNCATE TABLE` is not permitted for the `replication_connection_configuration` table.

The following table shows the correspondence between `replication_connection_configuration` columns and `SHOW REPLICA STATUS` columns.

`replication_connection_configuration` Column	**`SHOW REPLICA STATUS` Column**
`CHANNEL_NAME`	`Channel_name`
`HOST`	`Source_Host`
`PORT`	`Source_Port`
`USER`	`Source_User`
`NETWORK_INTERFACE`	`Source_Bind`
`AUTO_POSITION`	`Auto_Position`
`SSL_ALLOWED`	`Source_SSL_Allowed`
`SSL_CA_FILE`	`Source_SSL_CA_File`
`SSL_CA_PATH`	`Source_SSL_CA_Path`
`SSL_CERTIFICATE`	`Source_SSL_Cert`
`SSL_CIPHER`	`Source_SSL_Cipher`
`SSL_KEY`	`Source_SSL_Key`
`SSL_VERIFY_SERVER_CERTIFICATE`	`Source_SSL_Verify_Server_Cert`
`SSL_CRL_FILE`	`Source_SSL_Crl`
`SSL_CRL_PATH`	`Source_SSL_Crlpath`
`CONNECTION_RETRY_INTERVAL`	`Source_Connect_Retry`
`CONNECTION_RETRY_COUNT`	`Source_Retry_Count`
`HEARTBEAT_INTERVAL`	None
`TLS_VERSION`	`Source_TLS_Version`
`PUBLIC_KEY_PATH`	`Source_public_key_path`
`GET_PUBLIC_KEY`	`Get_source_public_key`
`NETWORK_NAMESPACE`	`Network_Namespace`

`replication_connection_configuration` Column	`SHOW REPLICA STATUS` Column
`COMPRESSION_ALGORITHM`	[None]
`ZSTD_COMPRESSION_LEVEL`	[None]
`GTID_ONLY`	[None]

10.11.11 The replication_connection_status Table

This table shows the current status of the I/O thread that handles the replica's connection to the source, information on the last transaction queued in the relay log, and information on the transaction currently being queued in the relay log.

Compared to the `replication_connection_configuration` table, `replication_connection_status` changes more frequently. It contains values that change during the connection, whereas `replication_connection_configuration` contains values which define how the replica connects to the source and that remain constant during the connection.

The `replication_connection_status` table has these columns:

- `CHANNEL_NAME`

  The replication channel which this row is displaying. There is always a default replication channel, and more replication channels can be added. See Replication Channels for more information.

- `GROUP_NAME`

  If this server is a member of a group, shows the name of the group the server belongs to.

- `SOURCE_UUID`

  The `server_uuid` value from the source.

- `THREAD_ID`

  The I/O thread ID.

- `SERVICE_STATE`

  `ON` (thread exists and is active or idle), `OFF` (thread no longer exists), or `CONNECTING` (thread exists and is connecting to the source).

- `RECEIVED_TRANSACTION_SET`

  The set of global transaction IDs (GTIDs) corresponding to all transactions received by this replica. Empty if GTIDs are not in use. See GTID Sets for more information.

- `LAST_ERROR_NUMBER`, `LAST_ERROR_MESSAGE`

  The error number and error message of the most recent error that caused the I/O thread to stop. An error number of 0 and message of the empty string mean “no error.” If the `LAST_ERROR_MESSAGE` value is not empty, the error values also appear in the replica's error log.

  Issuing `RESET MASTER` or `RESET REPLICA` resets the values shown in these columns.

- `LAST_ERROR_TIMESTAMP`

A timestamp in *'YYYY-MM-DD hh:mm:ss[.fraction]'* format that shows when the most recent I/O error took place.

- `LAST_HEARTBEAT_TIMESTAMP`

  A timestamp in *'YYYY-MM-DD hh:mm:ss[.fraction]'* format that shows when the most recent heartbeat signal was received by a replica.

- `COUNT_RECEIVED_HEARTBEATS`

  The total number of heartbeat signals that a replica received since the last time it was restarted or reset, or a `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement was issued.

- `LAST_QUEUED_TRANSACTION`

  The global transaction ID (GTID) of the last transaction that was queued to the relay log.

- `LAST_QUEUED_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP`

  A timestamp in *'YYYY-MM-DD hh:mm:ss[.fraction]'* format that shows when the last transaction queued in the relay log was committed on the original source.

- `LAST_QUEUED_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP`

  A timestamp in *'YYYY-MM-DD hh:mm:ss[.fraction]'* format that shows when the last transaction queued in the relay log was committed on the immediate source.

- `LAST_QUEUED_TRANSACTION_START_QUEUE_TIMESTAMP`

  A timestamp in *'YYYY-MM-DD hh:mm:ss[.fraction]'* format that shows when the last transaction was placed in the relay log queue by this I/O thread.

- `LAST_QUEUED_TRANSACTION_END_QUEUE_TIMESTAMP`

  A timestamp in *'YYYY-MM-DD hh:mm:ss[.fraction]'* format that shows when the last transaction was queued to the relay log files.

- `QUEUEING_TRANSACTION`

  The global transaction ID (GTID) of the currently queueing transaction in the relay log.

- `QUEUEING_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP`

  A timestamp in *'YYYY-MM-DD hh:mm:ss[.fraction]'* format that shows when the currently queueing transaction was committed on the original source.

- `QUEUEING_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP`

  A timestamp in *'YYYY-MM-DD hh:mm:ss[.fraction]'* format that shows when the currently queueing transaction was committed on the immediate source.

- `QUEUEING_TRANSACTION_START_QUEUE_TIMESTAMP`

  A timestamp in *'YYYY-MM-DD hh:mm:ss[.fraction]'* format that shows when the first event of the currently queueing transaction was written to the relay log by this I/O thread.

When the Performance Schema is disabled, local timing information is not collected, so the fields showing the start and end timestamps for queued transactions are zero.

The `replication_connection_status` table has these indexes:

- Primary key on (`CHANNEL_NAME`)
- Index on (`THREAD_ID`)

The following table shows the correspondence between `replication_connection_status` columns and `SHOW REPLICA STATUS` columns.

`replication_connection_status` Column	**`SHOW REPLICA STATUS` Column**
`SOURCE_UUID`	`Master_UUID`
`THREAD_ID`	None
`SERVICE_STATE`	`Replica_IO_Running`
`RECEIVED_TRANSACTION_SET`	`Retrieved_Gtid_Set`
`LAST_ERROR_NUMBER`	`Last_IO_Errno`
`LAST_ERROR_MESSAGE`	`Last_IO_Error`
`LAST_ERROR_TIMESTAMP`	`Last_IO_Error_Timestamp`

10.11.12 The replication_group_communication_information Table

This table shows group configuration options for the whole replication group. The table is available only when Group Replication is installed.

The `replication_group_communication_information` table has these columns:

- `WRITE_CONCURRENCY`

  The maximum number of consensus instances that the group can execute in parallel. The default value is 10. See Using Group Replication Group Write Consensus.

- `PROTOCOL_VERSION`

  The Group Replication communication protocol version, which determines what messaging capabilities are used. This is set to accommodate the oldest MySQL Server version that you want the group to support. See Setting a Group's Communication Protocol Version.

- `WRITE_CONSENSUS_LEADERS_PREFERRED`

  The leader or leaders that Group Replication has instructed the group communication engine to use to drive consensus. For a group in single-primary mode with the `group_replication_paxos_single_leader` system variable set to `ON` and the communication protocol version set to 8.0.27 or above, the single consensus leader is the group's primary. Otherwise, all group members are used as leaders, so they are all shown here. See Single Consensus Leader.

- `WRITE_CONSENSUS_LEADERS_ACTUAL`

  The actual leader or leader that the group communication engine is using to drive consensus. If a single consensus leader is in use for the group, and the primary is currently unhealthy, the group communication selects an alternative consensus leader. In this situation, the group member specified here can differ from the preferred group member.

- `WRITE_CONSENSUS_SINGLE_LEADER_CAPABLE`

  Whether the replication group is capable of using a single consensus leader. 1 means that the group was started with the use of a single leader enabled (`group_replication_paxos_single_leader`

= ON), and this is still shown if the value of group_replication_paxos_single_leader has since been changed on this group member. 0 means that the group was started with single leader mode disabled (group_replication_paxos_single_leader = OFF), or has a Group Replication communication protocol version that does not support the use of a single consensus leader (below 8.0.27). This information is only returned for group members in ONLINE or RECOVERING state.

The replication_group_communication_information table has no indexes.

TRUNCATE TABLE is not permitted for the replication_group_communication_information table.

10.11.13 The replication_group_configuration_version Table

This table displays the version of the member actions configuration for replication group members. The table is available only when Group Replication is installed. Whenever a member action is enabled or disabled using the group_replication_enable_member_action() and group_replication_disable_member_action() functions, the version number is incremented. You can reset the member actions configuration using the group_replication_reset_member_actions() function, which resets the member actions configuration to the default settings, and resets its version number to 1. For more information, see Configuring Member Actions.

The replication_group_configuration_version table has these columns:

- NAME

  The name of the configuration.

- VERSION

  The version number of the configuration.

The replication_group_configuration_version table has no indexes.

TRUNCATE TABLE is not permitted for the replication_group_configuration_version table.

10.11.14 The replication_group_member_actions Table

This table lists the member actions that are included in the member actions configuration for replication group members. The table is available only when Group Replication is installed. You can reset the member actions configuration using the group_replication_reset_member_actions() function. For more information, see Configuring Member Actions.

The replication_group_member_actions table has these columns:

- NAME

  The name of the member action.

- EVENT

  The event that triggers the member action.

- ENABLED

  Whether the member action is currently enabled. Member actions can be enabled using the group_replication_enable_member_action() function and disabled using the group_replication_disable_member_action() function.

- `TYPE`

  The type of member action. `INTERNAL` is an action that is provided by the Group Replication plugin.

- `PRIORITY`

  The priority of the member action. Actions with lower priority values are actioned first.

- `ERROR_HANDLING`

  The action that Group Replication takes if an error occurs when the member action is being carried out. `IGNORE` means that an error message is logged to say that the member action failed, but no further action is taken. `CRITICAL` means that the member moves into `ERROR` state, and takes the action specified by the `group_replication_exit_state_action` system variable.

The `replication_group_member_actions` table has no indexes.

`TRUNCATE TABLE` is not permitted for the `replication_group_member_actions` table.

10.11.15 The replication_group_member_stats Table

This table shows statistical information for replication group members. It is populated only when Group Replication is running.

The `replication_group_member_stats` table has these columns:

- `CHANNEL_NAME`

  Name of the Group Replication channel

- `VIEW_ID`

  Current view identifier for this group.

- `MEMBER_ID`

  The member server UUID. This has a different value for each member in the group. This also serves as a key because it is unique to each member.

- `COUNT_TRANSACTIONS_IN_QUEUE`

  The number of transactions in the queue pending conflict detection checks. Once the transactions have been checked for conflicts, if they pass the check, they are queued to be applied as well.

- `COUNT_TRANSACTIONS_CHECKED`

  The number of transactions that have been checked for conflicts.

- `COUNT_CONFLICTS_DETECTED`

  The number of transactions that have not passed the conflict detection check.

- `COUNT_TRANSACTIONS_ROWS_VALIDATING`

  Number of transaction rows which can be used for certification, but have not been garbage collected. Can be thought of as the current size of the conflict detection database against which each transaction is certified.

- `TRANSACTIONS_COMMITTED_ALL_MEMBERS`

  The transactions that have been successfully committed on all members of the replication group, shown as GTID Sets. This is updated at a fixed time interval.

- `LAST_CONFLICT_FREE_TRANSACTION`

  The transaction identifier of the last conflict free transaction which was checked.

- `COUNT_TRANSACTIONS_REMOTE_IN_APPLIER_QUEUE`

  The number of transactions that this member has received from the replication group which are waiting to be applied.

- `COUNT_TRANSACTIONS_REMOTE_APPLIED`

  Number of transactions this member has received from the group and applied.

- `COUNT_TRANSACTIONS_LOCAL_PROPOSED`

  Number of transactions which originated on this member and were sent to the group.

- `COUNT_TRANSACTIONS_LOCAL_ROLLBACK`

  Number of transactions which originated on this member and were rolled back by the group.

The `replication_group_member_stats` table has no indexes.

`TRUNCATE TABLE` is not permitted for the `replication_group_member_stats` table.

10.11.16 The replication_group_members Table

This table shows network and status information for replication group members. The network addresses shown are the addresses used to connect clients to the group, and should not be confused with the member's internal group communication address specified by `group_replication_local_address`.

The `replication_group_members` table has these columns:

- `CHANNEL_NAME`

  Name of the Group Replication channel.

- `MEMBER_ID`

  The member server UUID. This has a different value for each member in the group. This also serves as a key because it is unique to each member.

- `MEMBER_HOST`

  Network address of this member (host name or IP address). Retrieved from the member's `hostname` variable. This is the address which clients connect to, unlike the group_replication_local_address which is used for internal group communication.

- `MEMBER_PORT`

  Port on which the server is listening. Retrieved from the member's `port` variable.

- `MEMBER_STATE`

Current state of this member; can be any one of the following:

- `ONLINE`: The member is in a fully functioning state.
- `RECOVERING`: The server has joined a group from which it is retrieving data.
- `OFFLINE`: The group replication plugin is installed but has not been started.
- `ERROR`: The member has encountered an error, either during applying transactions or during the recovery phase, and is not participating in the group's transactions.
- `UNREACHABLE`: The failure detection process suspects that this member cannot be contacted, because the group messages have timed out.

See Group Replication Server States.

- `MEMBER_ROLE`

  Role of the member in the group, either `PRIMARY` or `SECONDARY`.

- `MEMBER_VERSION`

  MySQL version of the member.

- `MEMBER_COMMUNICATION_STACK`

  The communication stack used for the group, either the `XCOM` communication stack or the `MYSQL` communication stack.

  This column was added in MySQL 8.0.27.

The `replication_group_members` table has no indexes.

`TRUNCATE TABLE` is not permitted for the `replication_group_members` table.

10.12 Performance Schema NDB Cluster Tables

The following table shows all Performance Schema tables relating to the `NDBCLUSTER` storage engine.

Table 10.3 Performance Schema NDB Tables

Table Name	Description	Introduced
`ndb_sync_excluded_objects`	NDB objects which cannot be synchronized	8.0.21
`ndb_sync_pending_objects`	NDB objects waiting for synchronization	8.0.21

Beginning with NDB 8.0.16, automatic synchronization in `NDB` attempts to detect and synchronize automatically all mismatches in metadata between the NDB Cluster's internal dictionary and the MySQL Server's datadictionary. This is done by default in the background at regular intervals as determined by the `ndb_metadata_check_interval` system variable, unless disabled using `ndb_metadata_check` or overridden by setting `ndb_metadata_sync`. Prior to NDB 8.0.21, the only information readily accessible to users about this process was in the form of logging messages and object counts available (beginning with NDB 8.0.18) as the status variables `Ndb_metadata_detected_count`, `Ndb_metadata_synced_count`, and `Ndb_metadata_excluded_count` (prior to NDB 8.0.22, this

variable was named `Ndb_metadata_blacklist_size`). Beginning with NDB 8.0.21, more detailed information about the current state of automatic synchronization is exposed by a MySQL server acting as an SQL node in an NDB Cluster in these two Performance Schema tables:

- `ndb_sync_pending_objects`: Displays information about `NDB` database objects for which mismatches have been detected between the `NDB` dictionary and the MySQL data dictionary. When attempting to synchronize such objects, `NDB` removes the object from the queue awaiting synchronization, and from this table, and tries to reconcile the mismatch. If synchronization of the object fails due to a temporary error, it is picked up and added back to the queue (and to this table) the next time `NDB` performs mismatch detection; if the attempts fails due a permanent error, the object is added to the `ndb_sync_excluded_objects` table.

- `ndb_sync_excluded_objects`: Shows information about `NDB` database objects for which automatic synchronization has failed due to permanent errors resulting from mismatches which cannot be reconciled without manual intervention; these objects are blocklisted and not considered again for mismatch detection until this has been done.

The `ndb_sync_pending_objects` and `ndb_sync_excluded_objects` tables are present only if MySQL has support enabled for the `NDBCLUSTER` storage engine.

These tables are described in more detail in the following two sections.

10.12.1 The ndb_sync_pending_objects Table

This table provides information about `NDB` database objects for which mismatches have been detected and which are waiting to be synchronized between the `NDB` dictionary and the MySQL data dictionary.

Example information about `NDB` database objects awaiting synchronization:

```
mysql> SELECT * FROM performance_schema.ndb_sync_pending_objects;
+-------------+------+----------------+
| SCHEMA_NAME | NAME |  TYPE          |
+-------------+------+----------------+
| NULL        | lg1  |  LOGFILE GROUP |
| NULL        | ts1  |  TABLESPACE    |
| db1         | NULL |  SCHEMA        |
| test        | t1   |  TABLE         |
| test        | t2   |  TABLE         |
| test        | t3   |  TABLE         |
+-------------+------+----------------+
```

The `ndb_sync_pending_objects` table has these columns:

- `SCHEMA_NAME`: The name of the schema (database) in which the object awaiting synchronization resides; this is `NULL` for tablespaces and log file groups

- `NAME`: The name of the object awaiting synchronization; this is `NULL` if the object is a schema

- `TYPE`: The type of the object awaiting synchronization; this is one of `LOGFILE GROUP`, `TABLESPACE`, `SCHEMA`, or `TABLE`

The `ndb_sync_pending_objects` table was added in NDB 8.0.21.

10.12.2 The ndb_sync_excluded_objects Table

This table provides information about `NDB` database objects which cannot be automatically synchronized between NDB Cluster's dictionary and the MySQL data dictionary.

Example information about NDB database objects which cannot be synchronized with the MySQL data dictionary:

```
mysql> SELECT * FROM performance_schema.ndb_sync_excluded_objects\G
*************************** 1. row ***************************
SCHEMA_NAME: NULL
       NAME: lg1
       TYPE: LOGFILE GROUP
     REASON: Injected failure
*************************** 2. row ***************************
SCHEMA_NAME: NULL
       NAME: ts1
       TYPE: TABLESPACE
     REASON: Injected failure
*************************** 3. row ***************************
SCHEMA_NAME: db1
       NAME: NULL
       TYPE: SCHEMA
     REASON: Injected failure
*************************** 4. row ***************************
SCHEMA_NAME: test
       NAME: t1
       TYPE: TABLE
     REASON: Injected failure
*************************** 5. row ***************************
SCHEMA_NAME: test
       NAME: t2
       TYPE: TABLE
     REASON: Injected failure
*************************** 6. row ***************************
SCHEMA_NAME: test
       NAME: t3
       TYPE: TABLE
     REASON: Injected failure
```

The ndb_sync_excluded_objects table has these columns:

- SCHEMA_NAME: The name of the schema (database) in which the object which has failed to synchronize resides; this is NULL for tablespaces and log file groups
- NAME: The name of the object which has failed to synchronize; this is NULL if the object is a schema
- TYPE: The type of the object has failed to synchronize; this is one of LOGFILE GROUP, TABLESPACE, SCHEMA, or TABLE
- REASON: The reason for exclusion (blocklisting) of the object; that is, the reason for the failure to synchronize this object

  Possible reasons include the following:

  - Injected failure
  - Failed to determine if object existed in NDB
  - Failed to determine if object existed in DD
  - Failed to drop object in DD
  - Failed to get undofiles assigned to logfile group
  - Failed to get object id and version
  - Failed to install object in DD

- `Failed to get datafiles assigned to tablespace`
- `Failed to create schema`
- `Failed to determine if object was a local table`
- `Failed to invalidate table references`
- `Failed to set database name of NDB object`
- `Failed to get extra metadata of table`
- `Failed to migrate table with extra metadata version 1`
- `Failed to get object from DD`
- `Definition of table has changed in NDB Dictionary`
- `Failed to setup binlogging for table`

This list is not necessarily exhaustive, and is subject to change in future `NDB` releases.

The `ndb_sync_excluded_objects` table was added in NDB 8.0.21.

10.13 Performance Schema Lock Tables

The Performance Schema exposes lock information through these tables:

- `data_locks`: Data locks held and requested
- `data_lock_waits`: Relationships between data lock owners and data lock requestors blocked by those owners
- `metadata_locks`: Metadata locks held and requested
- `table_handles`: Table locks held and requested

The following sections describe these tables in more detail.

10.13.1 The data_locks Table

The `data_locks` table shows data locks held and requested. For information about which lock requests are blocked by which held locks, see Section 10.13.2, “The data_lock_waits Table”.

Example data lock information:

```
mysql> SELECT * FROM performance_schema.data_locks\G
*************************** 1. row ***************************
               ENGINE: INNODB
      ENGINE_LOCK_ID: 139664434886512:1059:139664350547912
ENGINE_TRANSACTION_ID: 2569
            THREAD_ID: 46
             EVENT_ID: 12
        OBJECT_SCHEMA: test
          OBJECT_NAME: t1
       PARTITION_NAME: NULL
    SUBPARTITION_NAME: NULL
           INDEX_NAME: NULL
```

```
OBJECT_INSTANCE_BEGIN: 139664350547912
            LOCK_TYPE: TABLE
            LOCK_MODE: IX
          LOCK_STATUS: GRANTED
            LOCK_DATA: NULL
*************************** 2. row ***************************
               ENGINE: INNODB
       ENGINE_LOCK_ID: 139664434886512:2:4:1:139664350544872
ENGINE_TRANSACTION_ID: 2569
            THREAD_ID: 46
             EVENT_ID: 12
        OBJECT_SCHEMA: test
          OBJECT_NAME: t1
       PARTITION_NAME: NULL
    SUBPARTITION_NAME: NULL
           INDEX_NAME: GEN_CLUST_INDEX
OBJECT_INSTANCE_BEGIN: 139664350544872
            LOCK_TYPE: RECORD
            LOCK_MODE: X
          LOCK_STATUS: GRANTED
            LOCK_DATA: supremum pseudo-record
```

Unlike most Performance Schema data collection, there are no instruments for controlling whether data lock information is collected or system variables for controlling data lock table sizes. The Performance Schema collects information that is already available in the server, so there is no memory or CPU overhead to generate this information or need for parameters that control its collection.

Use the `data_locks` table to help diagnose performance problems that occur during times of heavy concurrent load. For `InnoDB`, see the discussion of this topic at InnoDB INFORMATION_SCHEMA Transaction and Locking Information.

The `data_locks` table has these columns:

- `ENGINE`

  The storage engine that holds or requested the lock.

- `ENGINE_LOCK_ID`

  The ID of the lock held or requested by the storage engine. Tuples of (`ENGINE_LOCK_ID`, `ENGINE`) values are unique.

  Lock ID formats are internal and subject to change at any time. Applications should not rely on lock IDs having a particular format.

- `ENGINE_TRANSACTION_ID`

  The storage engine internal ID of the transaction that requested the lock. This can be considered the owner of the lock, although the lock might still be pending, not actually granted yet (`LOCK_STATUS='WAITING'`).

  If the transaction has not yet performed any write operation (is still considered read only), the column contains internal data that users should not try to interpret. Otherwise, the column is the transaction ID.

  For `InnoDB`, to obtain details about the transaction, join this column with the `TRX_ID` column of the `INFORMATION_SCHEMA INNODB_TRX` table.

- `THREAD_ID`

  The thread ID of the session that created the lock. To obtain details about the thread, join this column with the `THREAD_ID` column of the Performance Schema `threads` table.

THREAD_ID can be used together with EVENT_ID to determine the event during which the lock data structure was created in memory. (This event might have occurred before this particular lock request occurred, if the data structure is used to store multiple locks.)

- EVENT_ID

  The Performance Schema event that caused the lock. Tuples of (THREAD_ID, EVENT_ID) values implicitly identify a parent event in other Performance Schema tables:

  - The parent wait event in the events_waits_*xxx* tables
  - The parent stage event in the events_stages_*xxx* tables
  - The parent statement event in the events_statements_*xxx* tables
  - The parent transaction event in the events_transactions_current table

  To obtain details about the parent event, join the THREAD_ID and EVENT_ID columns with the columns of like name in the appropriate parent event table. See Section 14.2, “Obtaining Parent Event Information”.

- OBJECT_SCHEMA

  The schema that contains the locked table.

- OBJECT_NAME

  The name of the locked table.

- PARTITION_NAME

  The name of the locked partition, if any; NULL otherwise.

- SUBPARTITION_NAME

  The name of the locked subpartition, if any; NULL otherwise.

- INDEX_NAME

  The name of the locked index, if any; NULL otherwise.

  In practice, InnoDB always creates an index (GEN_CLUST_INDEX), so INDEX_NAME is non-NULL for InnoDB tables.

- OBJECT_INSTANCE_BEGIN

  The address in memory of the lock.

- LOCK_TYPE

  The type of lock.

  The value is storage engine dependent. For InnoDB, permitted values are RECORD for a row-level lock, TABLE for a table-level lock.

- LOCK_MODE

  How the lock is requested.

The value is storage engine dependent. For `InnoDB`, permitted values are `S[,GAP]`, `X[,GAP]`, `IS[,GAP]`, `IX[,GAP]`, `AUTO_INC`, and `UNKNOWN`. Lock modes other than `AUTO_INC` and `UNKNOWN` indicate gap locks, if present. For information about `S`, `X`, `IS`, `IX`, and gap locks, refer to InnoDB Locking.

- `LOCK_STATUS`

  The status of the lock request.

  The value is storage engine dependent. For `InnoDB`, permitted values are `GRANTED` (lock is held) and `WAITING` (lock is being waited for).

- `LOCK_DATA`

  The data associated with the lock, if any. The value is storage engine dependent. For `InnoDB`, a value is shown if the `LOCK_TYPE` is `RECORD`, otherwise the value is `NULL`. Primary key values of the locked record are shown for a lock placed on the primary key index. Secondary index values of the locked record are shown with primary key values appended for a lock placed on a secondary index. If there is no primary key, `LOCK_DATA` shows either the key values of a selected unique index or the unique `InnoDB` internal row ID number, according to the rules governing `InnoDB` clustered index use (see Clustered and Secondary Indexes). `LOCK_DATA` reports “supremum pseudo-record” for a lock taken on a supremum pseudo-record. If the page containing the locked record is not in the buffer pool because it was written to disk while the lock was held, `InnoDB` does not fetch the page from disk. Instead, `LOCK_DATA` reports `NULL`.

The `data_locks` table has these indexes:

- Primary key on (`ENGINE_LOCK_ID`, `ENGINE`)
- Index on (`ENGINE_TRANSACTION_ID`, `ENGINE`)
- Index on (`THREAD_ID`, `EVENT_ID`)
- Index on (`OBJECT_SCHEMA`, `OBJECT_NAME`, `PARTITION_NAME`, `SUBPARTITION_NAME`)

`TRUNCATE TABLE` is not permitted for the `data_locks` table.

> **Note**
>
> Prior to MySQL 8.0.1, information similar to that in the Performance Schema `data_locks` table is available in the `INFORMATION_SCHEMA.INNODB_LOCKS` table, which provides information about each lock that an `InnoDB` transaction has requested but not yet acquired, and each lock held by a transaction that is blocking another transaction. `INFORMATION_SCHEMA.INNODB_LOCKS` is deprecated and is removed as of MySQL 8.0.1. `data_locks` should be used instead.

Differences between `INNODB_LOCKS` and `data_locks`:

- If a transaction holds a lock, `INNODB_LOCKS` displays the lock only if another transaction is waiting for it. `data_locks` displays the lock regardless of whether any transaction is waiting for it.
- The `data_locks` table has no columns corresponding to `LOCK_SPACE`, `LOCK_PAGE`, or `LOCK_REC`.
- The `INNODB_LOCKS` table requires the global `PROCESS` privilege. The `data_locks` table requires the usual Performance Schema privilege of `SELECT` on the table to be selected from.

The following table shows the mapping from `INNODB_LOCKS` columns to `data_locks` columns. Use this information to migrate applications from one table to the other.

Table 10.4 Mapping from INNODB_LOCKS to data_locks Columns

INNODB_LOCKS Column	data_locks Column
LOCK_ID	ENGINE_LOCK_ID
LOCK_TRX_ID	ENGINE_TRANSACTION_ID
LOCK_MODE	LOCK_MODE
LOCK_TYPE	LOCK_TYPE
LOCK_TABLE (combined schema/table names)	OBJECT_SCHEMA (schema name), OBJECT_NAME (table name)
LOCK_INDEX	INDEX_NAME
LOCK_SPACE	None
LOCK_PAGE	None
LOCK_REC	None
LOCK_DATA	LOCK_DATA

10.13.2 The data_lock_waits Table

The data_lock_waits table implements a many-to-many relationship showing which data lock requests in the data_locks table are blocked by which held data locks in the data_locks table. Held locks in data_locks appear in data_lock_waits only if they block some lock request.

This information enables you to understand data lock dependencies between sessions. The table exposes not only which lock a session or transaction is waiting for, but which session or transaction currently holds that lock.

Example data lock wait information:

```
mysql> SELECT * FROM performance_schema.data_lock_waits\G
*************************** 1. row ***************************
                            ENGINE: INNODB
      REQUESTING_ENGINE_LOCK_ID: 140211201964816:2:4:2:140211086465800
REQUESTING_ENGINE_TRANSACTION_ID: 1555
            REQUESTING_THREAD_ID: 47
             REQUESTING_EVENT_ID: 5
REQUESTING_OBJECT_INSTANCE_BEGIN: 140211086465800
        BLOCKING_ENGINE_LOCK_ID: 140211201963888:2:4:2:140211086459880
  BLOCKING_ENGINE_TRANSACTION_ID: 1554
              BLOCKING_THREAD_ID: 46
               BLOCKING_EVENT_ID: 12
  BLOCKING_OBJECT_INSTANCE_BEGIN: 140211086459880
```

Unlike most Performance Schema data collection, there are no instruments for controlling whether data lock information is collected or system variables for controlling data lock table sizes. The Performance Schema collects information that is already available in the server, so there is no memory or CPU overhead to generate this information or need for parameters that control its collection.

Use the data_lock_waits table to help diagnose performance problems that occur during times of heavy concurrent load. For InnoDB, see the discussion of this topic at InnoDB INFORMATION_SCHEMA Transaction and Locking Information.

Because the columns in the data_lock_waits table are similar to those in the data_locks table, the column descriptions here are abbreviated. For more detailed column descriptions, see Section 10.13.1, “The data_locks Table”.

The `data_lock_waits` table has these columns:

- `ENGINE`

  The storage engine that requested the lock.

- `REQUESTING_ENGINE_LOCK_ID`

  The ID of the lock requested by the storage engine. To obtain details about the lock, join this column with the `ENGINE_LOCK_ID` column of the `data_locks` table.

- `REQUESTING_ENGINE_TRANSACTION_ID`

  The storage engine internal ID of the transaction that requested the lock.

- `REQUESTING_THREAD_ID`

  The thread ID of the session that requested the lock.

- `REQUESTING_EVENT_ID`

  The Performance Schema event that caused the lock request in the session that requested the lock.

- `REQUESTING_OBJECT_INSTANCE_BEGIN`

  The address in memory of the requested lock.

- `BLOCKING_ENGINE_LOCK_ID`

  The ID of the blocking lock. To obtain details about the lock, join this column with the `ENGINE_LOCK_ID` column of the `data_locks` table.

- `BLOCKING_ENGINE_TRANSACTION_ID`

  The storage engine internal ID of the transaction that holds the blocking lock.

- `BLOCKING_THREAD_ID`

  The thread ID of the session that holds the blocking lock.

- `BLOCKING_EVENT_ID`

  The Performance Schema event that caused the blocking lock in the session that holds it.

- `BLOCKING_OBJECT_INSTANCE_BEGIN`

  The address in memory of the blocking lock.

The `data_lock_waits` table has these indexes:

- Index on (`REQUESTING_ENGINE_LOCK_ID`, `ENGINE`)
- Index on (`BLOCKING_ENGINE_LOCK_ID`, `ENGINE`)
- Index on (`REQUESTING_ENGINE_TRANSACTION_ID`, `ENGINE`)
- Index on (`BLOCKING_ENGINE_TRANSACTION_ID`, `ENGINE`)
- Index on (`REQUESTING_THREAD_ID`, `REQUESTING_EVENT_ID`)

- Index on (BLOCKING_THREAD_ID, BLOCKING_EVENT_ID)

TRUNCATE TABLE is not permitted for the data_lock_waits table.

> **Note**
>
> Prior to MySQL 8.0.1, information similar to that in the Performance Schema data_lock_waits table is available in the INFORMATION_SCHEMA.INNODB_LOCK_WAITS table, which provides information about each blocked InnoDB transaction, indicating the lock it has requested and any locks that are blocking that request. INFORMATION_SCHEMA.INNODB_LOCK_WAITS is deprecated and is removed as of MySQL 8.0.1. data_lock_waits should be used instead.

The tables differ in the privileges required: The INNODB_LOCK_WAITS table requires the global PROCESS privilege. The data_lock_waits table requires the usual Performance Schema privilege of SELECT on the table to be selected from.

The following table shows the mapping from INNODB_LOCK_WAITS columns to data_lock_waits columns. Use this information to migrate applications from one table to the other.

Table 10.5 Mapping from INNODB_LOCK_WAITS to data_lock_waits Columns

INNODB_LOCK_WAITS Column	data_lock_waits Column
REQUESTING_TRX_ID	REQUESTING_ENGINE_TRANSACTION_ID
REQUESTED_LOCK_ID	REQUESTING_ENGINE_LOCK_ID
BLOCKING_TRX_ID	BLOCKING_ENGINE_TRANSACTION_ID
BLOCKING_LOCK_ID	BLOCKING_ENGINE_LOCK_ID

10.13.3 The metadata_locks Table

MySQL uses metadata locking to manage concurrent access to database objects and to ensure data consistency; see Metadata Locking. Metadata locking applies not just to tables, but also to schemas, stored programs (procedures, functions, triggers, scheduled events), tablespaces, user locks acquired with the GET_LOCK() function (see Locking Functions), and locks acquired with the locking service described in The Locking Service.

The Performance Schema exposes metadata lock information through the metadata_locks table:

- Locks that have been granted (shows which sessions own which current metadata locks).
- Locks that have been requested but not yet granted (shows which sessions are waiting for which metadata locks).
- Lock requests that have been killed by the deadlock detector.
- Lock requests that have timed out and are waiting for the requesting session's lock request to be discarded.

This information enables you to understand metadata lock dependencies between sessions. You can see not only which lock a session is waiting for, but which session currently holds that lock.

The metadata_locks table is read only and cannot be updated. It is autosized by default; to configure the table size, set the performance_schema_max_metadata_locks system variable at server startup.

Metadata lock instrumentation uses the `wait/lock/metadata/sql/mdl` instrument, which is enabled by default.

To control metadata lock instrumentation state at server startup, use lines like these in your `my.cnf` file:

- Enable:

```
[mysqld]
performance-schema-instrument='wait/lock/metadata/sql/mdl=ON'
```

- Disable:

```
[mysqld]
performance-schema-instrument='wait/lock/metadata/sql/mdl=OFF'
```

To control metadata lock instrumentation state at runtime, update the `setup_instruments` table:

- Enable:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'YES', TIMED = 'YES'
WHERE NAME = 'wait/lock/metadata/sql/mdl';
```

- Disable:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'NO', TIMED = 'NO'
WHERE NAME = 'wait/lock/metadata/sql/mdl';
```

The Performance Schema maintains `metadata_locks` table content as follows, using the `LOCK_STATUS` column to indicate the status of each lock:

- When a metadata lock is requested and obtained immediately, a row with a status of `GRANTED` is inserted.
- When a metadata lock is requested and not obtained immediately, a row with a status of `PENDING` is inserted.
- When a metadata lock previously requested is granted, its row status is updated to `GRANTED`.
- When a metadata lock is released, its row is deleted.
- When a pending lock request is canceled by the deadlock detector to break a deadlock (`ER_LOCK_DEADLOCK`), its row status is updated from `PENDING` to `VICTIM`.
- When a pending lock request times out (`ER_LOCK_WAIT_TIMEOUT`), its row status is updated from `PENDING` to `TIMEOUT`.
- When granted lock or pending lock request is killed, its row status is updated from `GRANTED` or `PENDING` to `KILLED`.
- The `VICTIM`, `TIMEOUT`, and `KILLED` status values are brief and signify that the lock row is about to be deleted.
- The `PRE_ACQUIRE_NOTIFY` and `POST_RELEASE_NOTIFY` status values are brief and signify that the metadata locking subsubsystem is notifying interested storage engines while entering lock acquisition operations or leaving lock release operations.

The `metadata_locks` table has these columns:

- `OBJECT_TYPE`

The type of lock used in the metadata lock subsystem. The value is one of `GLOBAL`, `SCHEMA`, `TABLE`, `FUNCTION`, `PROCEDURE`, `TRIGGER` (currently unused), `EVENT`, `COMMIT`, `USER LEVEL LOCK`, `TABLESPACE`, `BACKUP LOCK`, or `LOCKING SERVICE`.

A value of `USER LEVEL LOCK` indicates a lock acquired with `GET_LOCK()`. A value of `LOCKING SERVICE` indicates a lock acquired with the locking service described in The Locking Service.

- `OBJECT_SCHEMA`

  The schema that contains the object.

- `OBJECT_NAME`

  The name of the instrumented object.

- `OBJECT_INSTANCE_BEGIN`

  The address in memory of the instrumented object.

- `LOCK_TYPE`

  The lock type from the metadata lock subsystem. The value is one of `INTENTION_EXCLUSIVE`, `SHARED`, `SHARED_HIGH_PRIO`, `SHARED_READ`, `SHARED_WRITE`, `SHARED_UPGRADABLE`, `SHARED_NO_WRITE`, `SHARED_NO_READ_WRITE`, or `EXCLUSIVE`.

- `LOCK_DURATION`

  The lock duration from the metadata lock subsystem. The value is one of `STATEMENT`, `TRANSACTION`, or `EXPLICIT`. The `STATEMENT` and `TRANSACTION` values signify locks that are released implicitly at statement or transaction end, respectively. The `EXPLICIT` value signifies locks that survive statement or transaction end and are released by explicit action, such as global locks acquired with `FLUSH TABLES WITH READ LOCK`.

- `LOCK_STATUS`

  The lock status from the metadata lock subsystem. The value is one of `PENDING`, `GRANTED`, `VICTIM`, `TIMEOUT`, `KILLED`, `PRE_ACQUIRE_NOTIFY`, or `POST_RELEASE_NOTIFY`. The Performance Schema assigns these values as described previously.

- `SOURCE`

  The name of the source file containing the instrumented code that produced the event and the line number in the file at which the instrumentation occurs. This enables you to check the source to determine exactly what code is involved.

- `OWNER_THREAD_ID`

  The thread requesting a metadata lock.

- `OWNER_EVENT_ID`

  The event requesting a metadata lock.

The `metadata_locks` table has these indexes:

- Primary key on (`OBJECT_INSTANCE_BEGIN`)
- Index on (`OBJECT_TYPE`, `OBJECT_SCHEMA`, `OBJECT_NAME`)

- Index on (OWNER_THREAD_ID, OWNER_EVENT_ID)

TRUNCATE TABLE is not permitted for the metadata_locks table.

10.13.4 The table_handles Table

The Performance Schema exposes table lock information through the table_handles table to show the table locks currently in effect for each opened table handle. table_handles reports what is recorded by the table lock instrumentation. This information shows which table handles the server has open, how they are locked, and by which sessions.

The table_handles table is read only and cannot be updated. It is autosized by default; to configure the table size, set the performance_schema_max_table_handles system variable at server startup.

Table lock instrumentation uses the wait/lock/table/sql/handler instrument, which is enabled by default.

To control table lock instrumentation state at server startup, use lines like these in your my.cnf file:

- Enable:

```
[mysqld]
performance-schema-instrument='wait/lock/table/sql/handler=ON'
```

- Disable:

```
[mysqld]
performance-schema-instrument='wait/lock/table/sql/handler=OFF'
```

To control table lock instrumentation state at runtime, update the setup_instruments table:

- Enable:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'YES', TIMED = 'YES'
WHERE NAME = 'wait/lock/table/sql/handler';
```

- Disable:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'NO', TIMED = 'NO'
WHERE NAME = 'wait/lock/table/sql/handler';
```

The table_handles table has these columns:

- OBJECT_TYPE

  The table opened by a table handle.

- OBJECT_SCHEMA

  The schema that contains the object.

- OBJECT_NAME

  The name of the instrumented object.

- OBJECT_INSTANCE_BEGIN

  The table handle address in memory.

- `OWNER_THREAD_ID`

  The thread owning the table handle.

- `OWNER_EVENT_ID`

  The event which caused the table handle to be opened.

- `INTERNAL_LOCK`

  The table lock used at the SQL level. The value is one of `READ`, `READ WITH SHARED LOCKS`, `READ HIGH PRIORITY`, `READ NO INSERT`, `WRITE ALLOW WRITE`, `WRITE CONCURRENT INSERT`, `WRITE LOW PRIORITY`, or `WRITE`. For information about these lock types, see the `include/thr_lock.h` source file.

- `EXTERNAL_LOCK`

  The table lock used at the storage engine level. The value is one of `READ EXTERNAL` or `WRITE EXTERNAL`.

The `table_handles` table has these indexes:

- Primary key on (`OBJECT_INSTANCE_BEGIN`)
- Index on (`OBJECT_TYPE`, `OBJECT_SCHEMA`, `OBJECT_NAME`)
- Index on (`OWNER_THREAD_ID`, `OWNER_EVENT_ID`)

`TRUNCATE TABLE` is not permitted for the `table_handles` table.

10.14 Performance Schema System Variable Tables

The MySQL server maintains many system variables that indicate how it is configured (see Server System Variables). System variable information is available in these Performance Schema tables:

- `global_variables`: Global system variables. An application that wants only global values should use this table.
- `session_variables`: System variables for the current session. An application that wants all system variable values for its own session should use this table. It includes the session variables for its session, as well as the values of global variables that have no session counterpart.
- `variables_by_thread`: Session system variables for each active session. An application that wants to know the session variable values for specific sessions should use this table. It includes session variables only, identified by thread ID.
- `persisted_variables`: Provides a SQL interface to the `mysqld-auto.cnf` file that stores persisted global system variable settings. See Section 10.14.1, “Performance Schema persisted_variables Table”.
- `variables_info`: Shows, for each system variable, the source from which it was most recently set, and its range of values. See Section 10.14.2, “Performance Schema variables_info Table”.

The `SENSITIVE_VARIABLES_OBSERVER` privilege is required to view the values of sensitive system variables in these tables.

The session variable tables (`session_variables`, `variables_by_thread`) contain information only for active sessions, not terminated sessions.

The `global_variables` and `session_variables` tables have these columns:

- `VARIABLE_NAME`

  The system variable name.

- `VARIABLE_VALUE`

  The system variable value. For `global_variables`, this column contains the global value. For `session_variables`, this column contains the variable value in effect for the current session.

The `global_variables` and `session_variables` tables have these indexes:

- Primary key on (`VARIABLE_NAME`)

The `variables_by_thread` table has these columns:

- `THREAD_ID`

  The thread identifier of the session in which the system variable is defined.

- `VARIABLE_NAME`

  The system variable name.

- `VARIABLE_VALUE`

  The session variable value for the session named by the `THREAD_ID` column.

The `variables_by_thread` table has these indexes:

- Primary key on (`THREAD_ID`, `VARIABLE_NAME`)

The `variables_by_thread` table contains system variable information only about foreground threads. If not all threads are instrumented by the Performance Schema, this table misses some rows. In this case, the `Performance_schema_thread_instances_lost` status variable is greater than zero.

`TRUNCATE TABLE` is not supported for Performance Schema system variable tables.

10.14.1 Performance Schema persisted_variables Table

The `persisted_variables` table provides an SQL interface to the `mysqld-auto.cnf` file that stores persisted global system variable settings, enabling the file contents to be inspected at runtime using `SELECT` statements. Variables are persisted using `SET PERSIST` or `PERSIST_ONLY` statements; see SET Syntax for Variable Assignment. The table contains a row for each persisted system variable in the file. Variables not persisted do not appear in the table.

The `SENSITIVE_VARIABLES_OBSERVER` privilege is required to view the values of sensitive system variables in this table.

For information about persisted system variables, see Persisted System Variables.

Suppose that `mysqld-auto.cnf` looks like this (slightly reformatted):

```
{
  "Version": 1,
  "mysql_server": {
```

```
    "max_connections": {
      "Value": "1000",
      "Metadata": {
        "Timestamp": 1.519921706e+15,
        "User": "root",
        "Host": "localhost"
      }
    },
    "autocommit": {
      "Value": "ON",
      "Metadata": {
        "Timestamp": 1.519921707e+15,
        "User": "root",
        "Host": "localhost"
      }
    }
  }
}
```

Then `persisted_variables` has these contents:

```
mysql> SELECT * FROM performance_schema.persisted_variables;
+-----------------+----------------+
| VARIABLE_NAME   | VARIABLE_VALUE |
+-----------------+----------------+
| autocommit      | ON             |
| max_connections | 1000           |
+-----------------+----------------+
```

The `persisted_variables` table has these columns:

- `VARIABLE_NAME`

  The variable name listed in `mysqld-auto.cnf`.

- `VARIABLE_VALUE`

  The value listed for the variable in `mysqld-auto.cnf`.

`persisted_variables` has these indexes:

- Primary key on (`VARIABLE_NAME`)

`TRUNCATE TABLE` is not permitted for the `persisted_variables` table.

10.14.2 Performance Schema variables_info Table

The `variables_info` table shows, for each system variable, the source from which it was most recently set, and its range of values.

The `variables_info` table has these columns:

- `VARIABLE_NAME`

  The variable name.

- `VARIABLE_SOURCE`

  The source from which the variable was most recently set:

  - `COMMAND_LINE`

The variable was set on the command line.

- `COMPILED`

  The variable has its compiled-in default value. `COMPILED` is the value used for variables not set any other way.

- `DYNAMIC`

  The variable was set at runtime. This includes variables set within files specified using the `init_file` system variable.

- `EXPLICIT`

  The variable was set from an option file named with the `--defaults-file` option.

- `EXTRA`

  The variable was set from an option file named with the `--defaults-extra-file` option.

- `GLOBAL`

  The variable was set from a global option file. This includes option files not covered by `EXPLICIT`, `EXTRA`, `LOGIN`, `PERSISTED`, `SERVER`, or `USER`.

- `LOGIN`

  The variable was set from a user-specific login path file (`~/.mylogin.cnf`).

- `PERSISTED`

  The variable was set from a server-specific `mysqld-auto.cnf` option file. No row has this value if the server was started with `persisted_globals_load` disabled.

- `SERVER`

  The variable was set from a server-specific `$MYSQL_HOME/my.cnf` option file. For details about how `MYSQL_HOME` is set, see Using Option Files.

- `USER`

  The variable was set from a user-specific `~/.my.cnf` option file.

- `VARIABLE_PATH`

  If the variable was set from an option file, `VARIABLE_PATH` is the path name of that file. Otherwise, the value is the empty string.

- `MIN_VALUE`

  The minimum permitted value for the variable. For a variable whose type is not numeric, this is always 0.

- `MAX_VALUE`

  The maximum permitted value for the variable. For a variable whose type is not numeric, this is always 0.

- `SET_TIME`

The time at which the variable was most recently set. The default is the time at which the server initialized global system variables during startup.

- `SET_USER`, `SET_HOST`

  The user name and host name of the client user that most recently set the variable. If a client connects as `user17` from host `host34.example.com` using the account `'user17'@'%.example.com`, `SET_USER` and `SET_HOST` are `user17` and `host34.example.com`, respectively. For proxy user connections, these values correspond to the external (proxy) user, not the proxied user against which privilege checking is performed. The default for each column is the empty string, indicating that the variable has not been set since server startup.

The `variables_info` table has no indexes.

`TRUNCATE TABLE` is not permitted for the `variables_info` table.

If a variable with a `VARIABLE_SOURCE` value other than `DYNAMIC` is set at runtime, `VARIABLE_SOURCE` becomes `DYNAMIC` and `VARIABLE_PATH` becomes the empty string.

A system variable that has only a session value (such as `debug_sync`) cannot be set at startup or persisted. For session-only system variables, `VARIABLE_SOURCE` can be only `COMPILED` or `DYNAMIC`.

If a system variable has an unexpected `VARIABLE_SOURCE` value, consider your server startup method. For example, `mysqld_safe` reads option files and passes certain options it finds there as part of the command line that it uses to start `mysqld`. Consequently, some system variables that you set in option files might display in `variables_info` as `COMMAND_LINE`, rather than as `GLOBAL` or `SERVER` as you might otherwise expect.

Some sample queries that use the `variables_info` table, with representative output:

- Display variables set on the command line:

```
mysql> SELECT VARIABLE_NAME
       FROM performance_schema.variables_info
       WHERE VARIABLE_SOURCE = 'COMMAND_LINE'
       ORDER BY VARIABLE_NAME;
+---------------+
| VARIABLE_NAME |
+---------------+
| basedir       |
| datadir       |
| log_error     |
| pid_file      |
| plugin_dir    |
| port          |
+---------------+
```

- Display variables set from persistent storage:

```
mysql> SELECT VARIABLE_NAME
       FROM performance_schema.variables_info
       WHERE VARIABLE_SOURCE = 'PERSISTED'
       ORDER BY VARIABLE_NAME;
+--------------------------+
| VARIABLE_NAME            |
+--------------------------+
| event_scheduler          |
| max_connections          |
| validate_password.policy |
+--------------------------+
```

- Join `variables_info` with the `global_variables` table to display the current values of persisted variables, together with their range of values:

```
mysql> SELECT
         VI.VARIABLE_NAME, GV.VARIABLE_VALUE,
         VI.MIN_VALUE,VI.MAX_VALUE
       FROM performance_schema.variables_info AS VI
         INNER JOIN performance_schema.global_variables AS GV
         USING(VARIABLE_NAME)
       WHERE VI.VARIABLE_SOURCE = 'PERSISTED'
       ORDER BY VARIABLE_NAME;
+--------------------------+----------------+-----------+-----------+
| VARIABLE_NAME            | VARIABLE_VALUE | MIN_VALUE | MAX_VALUE |
+--------------------------+----------------+-----------+-----------+
| event_scheduler          | ON             | 0         | 0         |
| max_connections          | 200            | 1         | 100000    |
| validate_password.policy | STRONG         | 0         | 0         |
+--------------------------+----------------+-----------+-----------+
```

10.15 Performance Schema Status Variable Tables

The MySQL server maintains many status variables that provide information about its operation (see Server Status Variables). Status variable information is available in these Performance Schema tables:

- `global_status`: Global status variables. An application that wants only global values should use this table.
- `session_status`: Status variables for the current session. An application that wants all status variable values for its own session should use this table. It includes the session variables for its session, as well as the values of global variables that have no session counterpart.
- `status_by_thread`: Session status variables for each active session. An application that wants to know the session variable values for specific sessions should use this table. It includes session variables only, identified by thread ID.

There are also summary tables that provide status variable information aggregated by account, host name, and user name. See Section 10.20.12, “Status Variable Summary Tables”.

The session variable tables (`session_status`, `status_by_thread`) contain information only for active sessions, not terminated sessions.

The Performance Schema collects statistics for global status variables only for threads for which the `INSTRUMENTED` value is `YES` in the `threads` table. Statistics for session status variables are always collected, regardless of the `INSTRUMENTED` value.

The Performance Schema does not collect statistics for `Com_xxx` status variables in the status variable tables. To obtain global and per-session statement execution counts, use the `events_statements_summary_global_by_event_name` and `events_statements_summary_by_thread_by_event_name` tables, respectively. For example:

```
SELECT EVENT_NAME, COUNT_STAR
FROM performance_schema.events_statements_summary_global_by_event_name
WHERE EVENT_NAME LIKE 'statement/sql/%';
```

The `global_status` and `session_status` tables have these columns:

- `VARIABLE_NAME`

  The status variable name.

- `VARIABLE_VALUE`

  The status variable value. For `global_status`, this column contains the global value. For `session_status`, this column contains the variable value for the current session.

The `global_status` and `session_status` tables have these indexes:

- Primary key on (`VARIABLE_NAME`)

The `status_by_thread` table contains the status of each active thread. It has these columns:

- `THREAD_ID`

  The thread identifier of the session in which the status variable is defined.

- `VARIABLE_NAME`

  The status variable name.

- `VARIABLE_VALUE`

  The session variable value for the session named by the `THREAD_ID` column.

The `status_by_thread` table has these indexes:

- Primary key on (`THREAD_ID`, `VARIABLE_NAME`)

The `status_by_thread` table contains status variable information only about foreground threads. If the `performance_schema_max_thread_instances` system variable is not autoscaled (signified by a value of −1) and the maximum permitted number of instrumented thread objects is not greater than the number of background threads, the table is empty.

The Performance Schema supports `TRUNCATE TABLE` for status variable tables as follows:

- `global_status`: Resets thread, account, host, and user status. Resets global status variables except those that the server never resets.
- `session_status`: Not supported.
- `status_by_thread`: Aggregates status for all threads to the global status and account status, then resets thread status. If account statistics are not collected, the session status is added to host and user status, if host and user status are collected.

  Account, host, and user statistics are not collected if the `performance_schema_accounts_size`, `performance_schema_hosts_size`, and `performance_schema_users_size` system variables, respectively, are set to 0.

`FLUSH STATUS` adds the session status from all active sessions to the global status variables, resets the status of all active sessions, and resets account, host, and user status values aggregated from disconnected sessions.

10.16 Performance Schema Thread Pool Tables

> **Note**
>
> The Performance Schema tables described here are available as of MySQL 8.0.14. Prior to MySQL 8.0.14, use the corresponding `INFORMATION_SCHEMA` tables instead; see INFORMATION_SCHEMA Thread Pool Tables.

The following sections describe the Performance Schema tables associated with the thread pool plugin (see MySQL Enterprise Thread Pool). They provide information about thread pool operation:

- `tp_thread_group_state`: Information about thread pool thread group states.
- `tp_thread_group_stats`: Thread group statistics.
- `tp_thread_state`: 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 Performance Schema thread pool tables are implemented by the thread pool plugin and are loaded and unloaded when that plugin is loaded and unloaded (see Thread Pool Installation). No special configuration step for the tables is needed. However, the tables depend on the thread pool plugin being enabled. If the thread pool plugin is loaded but disabled, the tables are not created.

10.16.1 The tp_thread_group_state Table

> **Note**
>
> The Performance Schema table described here is available as of MySQL 8.0.14. Prior to MySQL 8.0.14, use the corresponding `INFORMATION_SCHEMA` table instead; see The INFORMATION_SCHEMA TP_THREAD_GROUP_STATE Table.

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.

The `tp_thread_group_state` table has these columns:

- `TP_GROUP_ID`

  The thread group ID. This is a unique key within the table.

- `CONSUMER_THREADS`

  The number of consumer threads. There is at most one thread ready to start executing if the active threads become stalled or blocked.

- `RESERVE_THREADS`

  The number of threads in the reserved state. This means that they are not started until there is a need to wake a new thread and there is no consumer thread. This is where most threads end up when the thread group has created more threads than needed for normal operation. Often a thread group needs additional threads for a short while and then does not need them again for a while. In this case, they go into the reserved state and remain until needed again. They take up some extra memory resources, but no extra computing resources.

- `CONNECT_THREAD_COUNT`

  The number of threads that are processing or waiting to process connection initialization and authentication. There can be a maximum of four connection threads per thread group; these threads expire after a period of inactivity.

- `CONNECTION_COUNT`

The number of connections using this thread group.

- `QUEUED_QUERIES`

  The number of statements waiting in the high-priority queue.

- `QUEUED_TRANSACTIONS`

  The number of statements waiting in the low-priority queue. These are the initial statements for transactions that have not started, so they also represent queued transactions.

- `STALL_LIMIT`

  The value of the `thread_pool_stall_limit` system variable for the thread group. This is the same value for all thread groups.

- `PRIO_KICKUP_TIMER`

  The value of the `thread_pool_prio_kickup_timer` system variable for the thread group. This is the same value for all thread groups.

- `ALGORITHM`

  The value of the `thread_pool_algorithm` system variable for the thread group. This is the same value for all thread groups.

- `THREAD_COUNT`

  The number of threads started in the thread pool as part of this thread group.

- `ACTIVE_THREAD_COUNT`

  The number of threads active in executing statements.

- `STALLED_THREAD_COUNT`

  The number of stalled statements in the thread group. A stalled statement could be executing, but from a thread pool perspective it is stalled and making no progress. A long-running statement quickly ends up in this category.

- `WAITING_THREAD_NUMBER`

  If there is a thread handling the polling of statements in the thread group, this specifies the thread number within this thread group. It is possible that this thread could be executing a statement.

- `OLDEST_QUEUED`

  How long in milliseconds the oldest queued statement has been waiting for execution.

- `MAX_THREAD_IDS_IN_GROUP`

  The maximum thread ID of the threads in the group. This is the same as `MAX(TP_THREAD_NUMBER)` for the threads when selected from the `tp_thread_state` table. That is, these two queries are equivalent:

```
SELECT TP_GROUP_ID, MAX_THREAD_IDS_IN_GROUP
FROM tp_thread_group_state;
SELECT TP_GROUP_ID, MAX(TP_THREAD_NUMBER)
FROM tp_thread_state GROUP BY TP_GROUP_ID;
```

The `tp_thread_group_state` table has these indexes:

- Unique index on (`TP_GROUP_ID`)

`TRUNCATE TABLE` is not permitted for the `tp_thread_group_state` table.

10.16.2 The tp_thread_group_stats Table

> **Note**
>
> The Performance Schema table described here is available as of MySQL 8.0.14. Prior to MySQL 8.0.14, use the corresponding `INFORMATION_SCHEMA` table instead; see The INFORMATION_SCHEMA TP_THREAD_GROUP_STATS Table.

The `tp_thread_group_stats` table reports statistics per thread group. There is one row per group.

The `tp_thread_group_stats` table has these columns:

- `TP_GROUP_ID`

  The thread group ID. This is a unique key within the table.

- `CONNECTIONS_STARTED`

  The number of connections started.

- `CONNECTIONS_CLOSED`

  The number of connections closed.

- `QUERIES_EXECUTED`

  The number of statements executed. This number is incremented when a statement starts executing, not when it finishes.

- `QUERIES_QUEUED`

  The number of statements received that were queued for execution. This does not count statements that the thread group was able to begin executing immediately without queuing, which can happen under the conditions described in Thread Pool Operation.

- `THREADS_STARTED`

  The number of threads started.

- `PRIO_KICKUPS`

  The number of statements that have been moved from low-priority queue to high-priority queue based on the value of the `thread_pool_prio_kickup_timer` system variable. If this number increases quickly, consider increasing the value of that variable. A quickly increasing counter means that the priority system is not keeping transactions from starting too early. For `InnoDB`, this most likely means deteriorating performance due to too many concurrent transactions..

- `STALLED_QUERIES_EXECUTED`

  The number of statements that have become defined as stalled due to executing for longer than the value of the `thread_pool_stall_limit` system variable.

- BECOME_CONSUMER_THREAD

  The number of times thread have been assigned the consumer thread role.

- BECOME_RESERVE_THREAD

  The number of times threads have been assigned the reserve thread role.

- BECOME_WAITING_THREAD

  The number of times threads have been assigned the waiter thread role. When statements are queued, this happens very often, even in normal operation, so rapid increases in this value are normal in the case of a highly loaded system where statements are queued up.

- WAKE_THREAD_STALL_CHECKER

  The number of times the stall check thread decided to wake or create a thread to possibly handle some statements or take care of the waiter thread role.

- SLEEP_WAITS

  The number of THD_WAIT_SLEEP waits. These occur when threads go to sleep (for example, by calling the SLEEP() function).

- DISK_IO_WAITS

  The number of THD_WAIT_DISKIO waits. These occur when threads perform disk I/O that is likely to not hit the file system cache. Such waits occur when the buffer pool reads and writes data to disk, not for normal reads from and writes to files.

- ROW_LOCK_WAITS

  The number of THD_WAIT_ROW_LOCK waits for release of a row lock by another transaction.

- GLOBAL_LOCK_WAITS

  The number of THD_WAIT_GLOBAL_LOCK waits for a global lock to be released.

- META_DATA_LOCK_WAITS

  The number of THD_WAIT_META_DATA_LOCK waits for a metadata lock to be released.

- TABLE_LOCK_WAITS

  The number of THD_WAIT_TABLE_LOCK waits for a table to be unlocked that the statement needs to access.

- USER_LOCK_WAITS

  The number of THD_WAIT_USER_LOCK waits for a special lock constructed by the user thread.

- BINLOG_WAITS

  The number of THD_WAIT_BINLOG_WAITS waits for the binary log to become free.

- GROUP_COMMIT_WAITS

  The number of THD_WAIT_GROUP_COMMIT waits. These occur when a group commit must wait for the other parties to complete their part of a transaction.

- FSYNC_WAITS

  The number of THD_WAIT_SYNC waits for a file sync operation.

The tp_thread_group_stats table has these indexes:

- Unique index on (TP_GROUP_ID)

TRUNCATE TABLE is not permitted for the tp_thread_group_stats table.

10.16.3 The tp_thread_state Table

> **Note**
>
> The Performance Schema table described here is available as of MySQL 8.0.14. Prior to MySQL 8.0.14, use the corresponding INFORMATION_SCHEMA table instead; see The INFORMATION_SCHEMA TP_THREAD_STATE Table.

The tp_thread_state table has one row per thread created by the thread pool to handle connections.

The tp_thread_state table has these columns:

- TP_GROUP_ID

  The thread group ID.

- TP_THREAD_NUMBER

  The ID of the thread within its thread group. TP_GROUP_ID and TP_THREAD_NUMBER together provide a unique key within the table.

- PROCESS_COUNT

  The 10ms interval in which the statement that uses this thread is currently executing. 0 means no statement is executing, 1 means it is in the first 10ms, and so forth.

- WAIT_TYPE

  The type of wait for the thread. NULL means the thread is not blocked. Otherwise, the thread is blocked by a call to thd_wait_begin() and the value specifies the type of wait. The *xxx*_WAIT columns of the tp_thread_group_stats table accumulate counts for each wait type.

  The WAIT_TYPE value is a string that describes the type of wait, as shown in the following table.

Table 10.6 tp_thread_state Table WAIT_TYPE Values

Wait Type	Meaning
THD_WAIT_SLEEP	Waiting for sleep
THD_WAIT_DISKIO	Waiting for Disk IO
THD_WAIT_ROW_LOCK	Waiting for row lock
THD_WAIT_GLOBAL_LOCK	Waiting for global lock
THD_WAIT_META_DATA_LOCK	Waiting for metadata lock
THD_WAIT_TABLE_LOCK	Waiting for table lock
THD_WAIT_USER_LOCK	Waiting for user lock

Wait Type	Meaning
THD_WAIT_BINLOG	Waiting for binlog
THD_WAIT_GROUP_COMMIT	Waiting for group commit
THD_WAIT_SYNC	Waiting for fsync

- TP_THREAD_TYPE

  The type of thread. The value shown in this column is one of CONNECTION_HANDLER_WORKER_THREAD, LISTENER_WORKER_THREAD, QUERY_WORKER_THREAD, or TIMER_WORKER_THREAD.

  This column was added in MySQL 8.0.32.

- THREAD_ID

  This thread's unique identifier. The value is the same as that used in the THREAD_ID column of the Performance Schema threads table.

  This column was added in MySQL 8.0.32.

The tp_thread_state table has these indexes:

- Unique index on (TP_GROUP_ID, TP_THREAD_NUMBER)

TRUNCATE TABLE is not permitted for the tp_thread_state table.

10.17 Performance Schema Firewall Tables

> **Note**
>
> The Performance Schema tables described here are available as of MySQL 8.0.23. Prior to MySQL 8.0.23, use the corresponding INFORMATION_SCHEMA tables instead; see MySQL Enterprise Firewall Tables.

The following sections describe the Performance Schema tables associated with MySQL Enterprise Firewall (see MySQL Enterprise Firewall). They provide information about firewall operation:

- firewall_groups: Information about firewall group profiles.
- firewall_group_allowlist: Allowlist rules of registered firewall group profiles.
- firewall_membership: Members (accounts) of registered firewall group profiles.

10.17.1 The firewall_groups Table

The firewall_groups table provides a view into the in-memory data cache for MySQL Enterprise Firewall. It lists names and operational modes of registered firewall group profiles. It is used in conjunction with the mysql.firewall_groups system table that provides persistent storage of firewall data; see MySQL Enterprise Firewall Tables.

The firewall_groups table has these columns:

- NAME

  The group profile name.

- `MODE`

  The current operational mode for the profile. Permitted mode values are `OFF`, `DETECTING`, `PROTECTING`, and `RECORDING`. For details about their meanings, see Firewall Concepts.

- `USERHOST`

  The training account for the group profile, to be used when the profile is in `RECORDING` mode. The value is `NULL`, or a non-`NULL` account that has the format *`user_name@host_name`*:

  - If the value is `NULL`, the firewall records allowlist rules for statements received from any account that is a member of the group.
  - If the value is non-`NULL`, the firewall records allowlist rules only for statements received from the named account (which should be a member of the group).

The `firewall_groups` table has no indexes.

`TRUNCATE TABLE` is not permitted for the `firewall_groups` table.

The `firewall_groups` table was added in MySQL 8.0.23.

10.17.2 The firewall_group_allowlist Table

The `firewall_group_allowlist` table provides a view into the in-memory data cache for MySQL Enterprise Firewall. It lists allowlist rules of registered firewall group profiles. It is used in conjunction with the `mysql.firewall_group_allowlist` system table that provides persistent storage of firewall data; see MySQL Enterprise Firewall Tables.

The `firewall_group_allowlist` table has these columns:

- `NAME`

  The group profile name.

- `RULE`

  A normalized statement indicating an acceptable statement pattern for the profile. A profile allowlist is the union of its rules.

The `firewall_group_allowlist` table has no indexes.

`TRUNCATE TABLE` is not permitted for the `firewall_group_allowlist` table.

The `firewall_group_allowlist` table was added in MySQL 8.0.23.

10.17.3 The firewall_membership Table

The `firewall_membership` table provides a view into the in-memory data cache for MySQL Enterprise Firewall. It lists the members (accounts) of registered firewall group profiles. It is used in conjunction with the `mysql.firewall_membership` system table that provides persistent storage of firewall data; see MySQL Enterprise Firewall Tables.

The `firewall_membership` table has these columns:

- `GROUP_ID`

  The group profile name.

- `MEMBER_ID`

  The name of an account that is a member of the profile.

The `firewall_membership` table has no indexes.

`TRUNCATE TABLE` is not permitted for the `firewall_membership` table.

The `firewall_membership` table was added in MySQL 8.0.23.

10.18 Performance Schema Keyring Tables

The following sections describe the Performance Schema tables associated with the MySQL keyring (see The MySQL Keyring). They provide information about keyring operation:

- `keyring_component_status`: Information about the keyring component in use.
- `keyring_keys`: Metadata for keys in the MySQL keyring.

10.18.1 The keyring_component_status Table

The `keyring_component_status` table (available as of MySQL 8.0.24) provides status information about the properties of the keyring component in use, if one is installed. The table is empty if no keyring component is installed (for example, if the keyring is not being used, or is configured to manage the keystore using a keyring plugin rather than a keyring component).

There is no fixed set of properties. Each keyring component is free to define its own set.

Example `keyring_component_status` contents:

```
mysql> SELECT * FROM performance_schema.keyring_component_status;
+--------------------+-------------------------------------------------+
| STATUS_KEY         | STATUS_VALUE                                    |
+--------------------+-------------------------------------------------+
| Component_name     | component_keyring_file                          |
| Author             | Oracle Corporation                              |
| License            | GPL                                             |
| Implementation_name | component_keyring_file                         |
| Version            | 1.0                                             |
| Component_status   | Active                                          |
| Data_file          | /usr/local/mysql/keyring/component_keyring_file |
| Read_only          | No                                              |
+--------------------+-------------------------------------------------+
```

The `keyring_component_status` table has these columns:

- `STATUS_KEY`

  The status item name.

- `STATUS_VALUE`

  The status item value.

The `keyring_component_status` table has no indexes.

`TRUNCATE TABLE` is not permitted for the `keyring_component_status` table.

10.18.2 The keyring_keys table

MySQL Server supports a keyring that enables internal server components and plugins to securely store sensitive information for later retrieval. See The MySQL Keyring.

As of MySQL 8.0.16, the `keyring_keys` table exposes metadata for keys in the keyring. Key metadata includes key IDs, key owners, and backend key IDs. The `keyring_keys` table does *not* expose any sensitive keyring data such as key contents.

The `keyring_keys` table has these columns:

- `KEY_ID`

  The key identifier.

- `KEY_OWNER`

  The owner of the key.

- `BACKEND_KEY_ID`

  The ID used for the key by the keyring backend.

The `keyring_keys` table has no indexes.

`TRUNCATE TABLE` is not permitted for the `keyring_keys` table.

10.19 Performance Schema Clone Tables

> **Note**
>
> The Performance Schema tables described here are available as of MySQL 8.0.17.

The following sections describe the Performance Schema tables associated with the clone plugin (see The Clone Plugin). The tables provide information about cloning operations.

- `clone_status`: status information about the current or last executed cloning operation.
- `clone_progress`: progress information about the current or last executed cloning operation.

The Performance Schema clone tables are implemented by the clone plugin and are loaded and unloaded when that plugin is loaded and unloaded (see Installing the Clone Plugin). No special configuration step for the tables is needed. However, the tables depend on the clone plugin being enabled. If the clone plugin is loaded but disabled, the tables are not created.

The Performance Schema clone plugin tables are used only on the recipient MySQL server instance. The data is persisted across server shutdown and restart.

10.19.1 The clone_status Table

> **Note**
>
> The Performance Schema table described here is available as of MySQL 8.0.17.

The `clone_status` table shows the status of the current or last executed cloning operation only. The table only ever contains one row of data, or is empty.

The `clone_status` table has these columns:

- `ID`

A unique cloning operation identifier in the current MySQL server instance.

- `PID`

  Process list ID of the session executing the cloning operation.

- `STATE`

  Current state of the cloning operation. Values include `Not Started`, `In Progress`, `Completed`, and `Failed`.

- `BEGIN_TIME`

  A timestamp in *`'YYYY-MM-DD hh:mm:ss[.fraction]'`* format that shows when the cloning operation started.

- `END_TIME`

  A timestamp in *`'YYYY-MM-DD hh:mm:ss[.fraction]'`* format that shows when the cloning operation finished. Reports NULL if the operation has not ended.

- `SOURCE`

  The donor MySQL server address in '`HOST:PORT`' format. The column displays '`LOCAL INSTANCE`' for a local cloning operation.

- `DESTINATION`

  The directory being cloned to.

- `ERROR_NO`

  The error number reported for a failed cloning operation.

- `ERROR_MESSAGE`

  The error message string for a failed cloning operation.

- `BINLOG_FILE`

  The name of the binary log file up to which data is cloned.

- `BINLOG_POSITION`

  The binary log file offset up to which data is cloned.

- `GTID_EXECUTED`

  The GTID value for the last cloned transaction.

The `clone_status` table is read-only. DDL, including `TRUNCATE TABLE`, is not permitted.

10.19.2 The clone_progress Table

> **Note**
>
> The Performance Schema table described here is available as of MySQL 8.0.17.

The `clone_progress` table shows progress information for the current or last executed cloning operation only.

The stages of a cloning operation include `DROP DATA`, `FILE COPY`, `PAGE_COPY`, `REDO_COPY`, `FILE_SYNC`, `RESTART`, and `RECOVERY`. A cloning operation produces a record for each stage. The table therefore only ever contains seven rows of data, or is empty.

The `clone_progress` table has these columns:

- `ID`

  A unique cloning operation identifier in the current MySQL server instance.

- `STAGE`

  The name of the current cloning stage. Stages include `DROP DATA`, `FILE COPY`, `PAGE_COPY`, `REDO_COPY`, `FILE_SYNC`, `RESTART`, and `RECOVERY`.

- `STATE`

  The current state of the cloning stage. States include `Not Started`, `In Progress`, and `Completed`.

- `BEGIN_TIME`

  A timestamp in *`'YYYY-MM-DD hh:mm:ss[.fraction]'`* format that shows when the cloning stage started. Reports NULL if the stage has not started.

- `END_TIME`

  A timestamp in *`'YYYY-MM-DD hh:mm:ss[.fraction]'`* format that shows when the cloning stage finished. Reports NULL if the stage has not ended.

- `THREADS`

  The number of concurrent threads used in the stage.

- `ESTIMATE`

  The estimated amount of data for the current stage, in bytes.

- `DATA`

  The amount of data transferred in current state, in bytes.

- `NETWORK`

  The amount of network data transferred in the current state, in bytes.

- `DATA_SPEED`

  The current actual speed of data transfer, in bytes per second. This value may differ from the requested maximum data transfer rate defined by `clone_max_data_bandwidth`.

- `NETWORK_SPEED`

  The current speed of network transfer in bytes per second.

The `clone_progress` table is read-only. DDL, including `TRUNCATE TABLE`, is not permitted.

10.20 Performance Schema Summary Tables

Summary tables provide aggregated information for terminated events over time. The tables in this group summarize event data in different ways.

Each summary table has grouping columns that determine how to group the data to be aggregated, and summary columns that contain the aggregated values. Tables that summarize events in similar ways often have similar sets of summary columns and differ only in the grouping columns used to determine how events are aggregated.

Summary tables can be truncated with `TRUNCATE TABLE`. 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.

Wait Event Summaries

Table 10.7 Performance Schema Wait Event Summary Tables

Table Name	Description
`events_waits_summary_by_account_by_event_name`	Wait events per account and event name
`events_waits_summary_by_host_by_event_name`	Wait events per host name and event name
`events_waits_summary_by_instance`	Wait events per instance
`events_waits_summary_by_thread_by_event_name`	Wait events per thread and event name
`events_waits_summary_by_user_by_event_name`	Wait events per user name and event name
`events_waits_summary_global_by_event_name`	Wait events per event name

Stage Summaries

Table 10.8 Performance Schema Stage Event Summary Tables

Table Name	Description
`events_stages_summary_by_account_by_event_name`	Stage events per account and event name
`events_stages_summary_by_host_by_event_name`	Stage events per host name and event name
`events_stages_summary_by_thread_by_event_name`	Stage waits per thread and event name
`events_stages_summary_by_user_by_event_name`	Stage events per user name and event name
`events_stages_summary_global_by_event_name`	Stage waits per event name

Statement Summaries

Table 10.9 Performance Schema Statement Event Summary Tables

Table Name	Description
`events_statements_histogram_by_digest`	Statement histograms per schema and digest value
`events_statements_histogram_global`	Statement histogram summarized globally
`events_statements_summary_by_account_by_event_name`	Statement events per account and event name
`events_statements_summary_by_digest`	Statement events per schema and digest value
`events_statements_summary_by_host_by_event_name`	Statement events per host name and event name
`events_statements_summary_by_program`	Statement events per stored program

Table Name	Description
events_statements_summary_by_thread_by_event_name	Statement events per thread and event name
events_statements_summary_by_user_by_event_name	Statement events per user name and event name
events_statements_summary_global_by_event_name	Statement events per event name
prepared_statements_instances	Prepared statement instances and statistics

Transaction Summaries

Table 10.10 Performance Schema Transaction Event Summary Tables

Table Name	Description
events_transactions_summary_by_account_by_event_name	Transaction events per account and event name
events_transactions_summary_by_host_by_event_name	Transaction events per host name and event name
events_transactions_summary_by_thread_by_event_name	Transaction events per thread and event name
events_transactions_summary_by_user_by_event_name	Transaction events per user name and event name
events_transactions_summary_global_by_event_name	Transaction events per event name

Object Wait Summaries

Table 10.11 Performance Schema Object Event Summary Tables

Table Name	Description
objects_summary_global_by_type	Object summaries

File I/O Summaries

Table 10.12 Performance Schema File I/O Event Summary Tables

Table Name	Description
file_summary_by_event_name	File events per event name
file_summary_by_instance	File events per file instance

Table I/O and Lock Wait Summaries

Table 10.13 Performance Schema Table I/O and Lock Wait Event Summary Tables

Table Name	Description
table_io_waits_summary_by_index_usage	Table I/O waits per index
table_io_waits_summary_by_table	Table I/O waits per table
table_lock_waits_summary_by_table	Table lock waits per table

Socket Summaries

Table 10.14 Performance Schema Socket Event Summary Tables

Table Name	Description
socket_summary_by_event_name	Socket waits and I/O per event name
socket_summary_by_instance	Socket waits and I/O per instance

Memory Summaries

Table 10.15 Performance Schema Memory Operation Summary Tables

Table Name	Description
memory_summary_by_account_by_event_name	Memory operations per account and event name
memory_summary_by_host_by_event_name	Memory operations per host and event name
memory_summary_by_thread_by_event_name	Memory operations per thread and event name
memory_summary_by_user_by_event_name	Memory operations per user and event name
memory_summary_global_by_event_name	Memory operations globally per event name

Error Summaries

Table 10.16 Performance Schema Error Summary Tables

Table Name	Description
events_errors_summary_by_account_by_error	Errors per account and error code
events_errors_summary_by_host_by_error	Errors per host and error code
events_errors_summary_by_thread_by_error	Errors per thread and error code
events_errors_summary_by_user_by_error	Errors per user and error code
events_errors_summary_global_by_error	Errors per error code

Status Variable Summaries

Table 10.17 Performance Schema Error Status Variable Summary Tables

Table Name	Description
status_by_account	Session status variables per account
status_by_host	Session status variables per host name
status_by_user	Session status variables per user name

10.20.1 Wait Event Summary Tables

The Performance Schema maintains tables for collecting current and recent wait events, and aggregates that information in summary tables. Section 10.4, “Performance Schema Wait Event Tables” describes the events on which wait summaries are based. See that discussion for information about the content of wait events, the current and recent wait event tables, and how to control wait event collection, which is disabled by default.

Example wait event summary information:

```
mysql> SELECT *
       FROM performance_schema.events_waits_summary_global_by_event_name\G
...
*************************** 6. row ***************************
    EVENT_NAME: wait/synch/mutex/sql/BINARY_LOG::LOCK_index
    COUNT_STAR: 8
SUM_TIMER_WAIT: 2119302
MIN_TIMER_WAIT: 196092
AVG_TIMER_WAIT: 264912
MAX_TIMER_WAIT: 569421
...
```

```
*************************** 9. row ***************************
    EVENT_NAME: wait/synch/mutex/sql/hash_filo::lock
    COUNT_STAR: 69
SUM_TIMER_WAIT: 16848828
MIN_TIMER_WAIT: 0
AVG_TIMER_WAIT: 244185
MAX_TIMER_WAIT: 735345
...
```

Each wait event summary table has one or more grouping columns to indicate how the table aggregates events. Event names refer to names of event instruments in the `setup_instruments` table:

- `events_waits_summary_by_account_by_event_name` has `EVENT_NAME`, `USER`, and `HOST` columns. Each row summarizes events for a given account (user and host combination) and event name.
- `events_waits_summary_by_host_by_event_name` has `EVENT_NAME` and `HOST` columns. Each row summarizes events for a given host and event name.
- `events_waits_summary_by_instance` has `EVENT_NAME` and `OBJECT_INSTANCE_BEGIN` columns. Each row summarizes events for a given event name and object. If an instrument is used to create multiple instances, each instance has a unique `OBJECT_INSTANCE_BEGIN` value and is summarized separately in this table.
- `events_waits_summary_by_thread_by_event_name` has `THREAD_ID` and `EVENT_NAME` columns. Each row summarizes events for a given thread and event name.
- `events_waits_summary_by_user_by_event_name` has `EVENT_NAME` and `USER` columns. Each row summarizes events for a given user and event name.
- `events_waits_summary_global_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name. An instrument might be used to create multiple instances of the instrumented object. For example, if there is an instrument for a mutex that is created for each connection, there are as many instances as there are connections. The summary row for the instrument summarizes over all these instances.

Each wait event summary table has these summary columns containing aggregated values:

- `COUNT_STAR`

  The number of summarized events. This value includes all events, whether timed or nontimed.

- `SUM_TIMER_WAIT`

  The total wait time of the summarized timed events. This value is calculated only for timed events because nontimed events have a wait time of `NULL`. The same is true for the other *`xxx`*`_TIMER_WAIT` values.

- `MIN_TIMER_WAIT`

  The minimum wait time of the summarized timed events.

- `AVG_TIMER_WAIT`

  The average wait time of the summarized timed events.

- `MAX_TIMER_WAIT`

  The maximum wait time of the summarized timed events.

The wait event summary tables have these indexes:

- `events_waits_summary_by_account_by_event_name`:
  - Primary key on (`USER`, `HOST`, `EVENT_NAME`)
- `events_waits_summary_by_host_by_event_name`:
  - Primary key on (`HOST`, `EVENT_NAME`)
- `events_waits_summary_by_instance`:
  - Primary key on (`OBJECT_INSTANCE_BEGIN`)
  - Index on (`EVENT_NAME`)
- `events_waits_summary_by_thread_by_event_name`:
  - Primary key on (`THREAD_ID`, `EVENT_NAME`)
- `events_waits_summary_by_user_by_event_name`:
  - Primary key on (`USER`, `EVENT_NAME`)
- `events_waits_summary_global_by_event_name`:
  - Primary key on (`EVENT_NAME`)

`TRUNCATE TABLE` is permitted for wait summary tables. It has these effects:

- For summary tables not aggregated by account, host, or user, truncation resets the summary columns to zero rather than removing rows.
- For summary tables aggregated by account, host, or user, truncation removes rows for accounts, hosts, or users with no connections, and resets the summary columns to zero for the remaining rows.

In addition, each wait summary table that is aggregated by account, host, user, or thread is implicitly truncated by truncation of the connection table on which it depends, or truncation of `events_waits_summary_global_by_event_name`. For details, see Section 10.8, “Performance Schema Connection Tables”.

10.20.2 Stage Summary Tables

The Performance Schema maintains tables for collecting current and recent stage events, and aggregates that information in summary tables. Section 10.5, “Performance Schema Stage Event Tables” describes the events on which stage summaries are based. See that discussion for information about the content of stage events, the current and historical stage event tables, and how to control stage event collection, which is disabled by default.

Example stage event summary information:

```
mysql> SELECT *
       FROM performance_schema.events_stages_summary_global_by_event_name\G
...
*************************** 5. row ***************************
    EVENT_NAME: stage/sql/checking permissions
    COUNT_STAR: 57
SUM_TIMER_WAIT: 26501888880
MIN_TIMER_WAIT: 7317456
```

```
AVG_TIMER_WAIT: 464945295
MAX_TIMER_WAIT: 12858936792
...
*************************** 9. row ***************************
    EVENT_NAME: stage/sql/closing tables
    COUNT_STAR: 37
SUM_TIMER_WAIT: 662606568
MIN_TIMER_WAIT: 1593864
AVG_TIMER_WAIT: 17907891
MAX_TIMER_WAIT: 437977248
...
```

Each stage summary table has one or more grouping columns to indicate how the table aggregates events. Event names refer to names of event instruments in the `setup_instruments` table:

- `events_stages_summary_by_account_by_event_name` has `EVENT_NAME`, `USER`, and `HOST` columns. Each row summarizes events for a given account (user and host combination) and event name.
- `events_stages_summary_by_host_by_event_name` has `EVENT_NAME` and `HOST` columns. Each row summarizes events for a given host and event name.
- `events_stages_summary_by_thread_by_event_name` has `THREAD_ID` and `EVENT_NAME` columns. Each row summarizes events for a given thread and event name.
- `events_stages_summary_by_user_by_event_name` has `EVENT_NAME` and `USER` columns. Each row summarizes events for a given user and event name.
- `events_stages_summary_global_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name.

Each stage summary table has these summary columns containing aggregated values: `COUNT_STAR`, `SUM_TIMER_WAIT`, `MIN_TIMER_WAIT`, `AVG_TIMER_WAIT`, and `MAX_TIMER_WAIT`. These columns are analogous to the columns of the same names in the wait event summary tables (see Section 10.20.1, “Wait Event Summary Tables”), except that the stage summary tables aggregate events from `events_stages_current` rather than `events_waits_current`.

The stage summary tables have these indexes:

- `events_stages_summary_by_account_by_event_name`:
  - Primary key on (`USER`, `HOST`, `EVENT_NAME`)
- `events_stages_summary_by_host_by_event_name`:
  - Primary key on (`HOST`, `EVENT_NAME`)
- `events_stages_summary_by_thread_by_event_name`:
  - Primary key on (`THREAD_ID`, `EVENT_NAME`)
- `events_stages_summary_by_user_by_event_name`:
  - Primary key on (`USER`, `EVENT_NAME`)
- `events_stages_summary_global_by_event_name`:
  - Primary key on (`EVENT_NAME`)

`TRUNCATE TABLE` is permitted for stage summary tables. It has these effects:

- For summary tables not aggregated by account, host, or user, truncation resets the summary columns to zero rather than removing rows.
- For summary tables aggregated by account, host, or user, truncation removes rows for accounts, hosts, or users with no connections, and resets the summary columns to zero for the remaining rows.

In addition, each stage summary table that is aggregated by account, host, user, or thread is implicitly truncated by truncation of the connection table on which it depends, or truncation of `events_stages_summary_global_by_event_name`. For details, see Section 10.8, “Performance Schema Connection Tables”.

10.20.3 Statement Summary Tables

The Performance Schema maintains tables for collecting current and recent statement events, and aggregates that information in summary tables. Section 10.6, “Performance Schema Statement Event Tables” describes the events on which statement summaries are based. See that discussion for information about the content of statement events, the current and historical statement event tables, and how to control statement event collection, which is partially disabled by default.

Example statement event summary information:

```
mysql> SELECT *
       FROM performance_schema.events_statements_summary_global_by_event_name\G
*************************** 1. row ***************************
                EVENT_NAME: statement/sql/select
                COUNT_STAR: 54
            SUM_TIMER_WAIT: 38860400000
            MIN_TIMER_WAIT: 52400000
            AVG_TIMER_WAIT: 719600000
            MAX_TIMER_WAIT: 12631800000
             SUM_LOCK_TIME: 88000000
                SUM_ERRORS: 0
              SUM_WARNINGS: 0
         SUM_ROWS_AFFECTED: 0
             SUM_ROWS_SENT: 60
         SUM_ROWS_EXAMINED: 120
SUM_CREATED_TMP_DISK_TABLES: 0
     SUM_CREATED_TMP_TABLES: 21
       SUM_SELECT_FULL_JOIN: 16
 SUM_SELECT_FULL_RANGE_JOIN: 0
           SUM_SELECT_RANGE: 0
     SUM_SELECT_RANGE_CHECK: 0
            SUM_SELECT_SCAN: 41
      SUM_SORT_MERGE_PASSES: 0
             SUM_SORT_RANGE: 0
              SUM_SORT_ROWS: 0
              SUM_SORT_SCAN: 0
          SUM_NO_INDEX_USED: 22
     SUM_NO_GOOD_INDEX_USED: 0
                SUM_CPU_TIME: 0
      MAX_CONTROLLED_MEMORY: 2028360
           MAX_TOTAL_MEMORY: 2853429
            COUNT_SECONDARY: 0
...
```

Each statement summary table has one or more grouping columns to indicate how the table aggregates events. Event names refer to names of event instruments in the `setup_instruments` table:

- `events_statements_summary_by_account_by_event_name` has `EVENT_NAME`, `USER`, and `HOST` columns. Each row summarizes events for a given account (user and host combination) and event name.

- `events_statements_summary_by_digest` has `SCHEMA_NAME` and `DIGEST` columns. Each row summarizes events per schema and digest value. (The `DIGEST_TEXT` column contains the corresponding normalized statement digest text, but is neither a grouping nor a summary column. The `QUERY_SAMPLE_TEXT`, `QUERY_SAMPLE_SEEN`, and `QUERY_SAMPLE_TIMER_WAIT` columns also are neither grouping nor summary columns; they support statement sampling.)

  The maximum number of rows in the table is autosized at server startup. To set this maximum explicitly, set the `performance_schema_digests_size` system variable at server startup.

- `events_statements_summary_by_host_by_event_name` has `EVENT_NAME` and `HOST` columns. Each row summarizes events for a given host and event name.

- `events_statements_summary_by_program` has `OBJECT_TYPE`, `OBJECT_SCHEMA`, and `OBJECT_NAME` columns. Each row summarizes events for a given stored program (stored procedure or function, trigger, or event).

- `events_statements_summary_by_thread_by_event_name` has `THREAD_ID` and `EVENT_NAME` columns. Each row summarizes events for a given thread and event name.

- `events_statements_summary_by_user_by_event_name` has `EVENT_NAME` and `USER` columns. Each row summarizes events for a given user and event name.

- `events_statements_summary_global_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name.

- `prepared_statements_instances` has an `OBJECT_INSTANCE_BEGIN` column. Each row summarizes events for a given prepared statement.

Each statement summary table has these summary columns containing aggregated values (with exceptions as noted):

- `COUNT_STAR`, `SUM_TIMER_WAIT`, `MIN_TIMER_WAIT`, `AVG_TIMER_WAIT`, `MAX_TIMER_WAIT`

  These columns are analogous to the columns of the same names in the wait event summary tables (see Section 10.20.1, “Wait Event Summary Tables”), except that the statement summary tables aggregate events from `events_statements_current` rather than `events_waits_current`.

  The `prepared_statements_instances` table does not have these columns.

- `SUM_`*`xxx`*

  The aggregate of the corresponding *`xxx`* column in the `events_statements_current` table. For example, the `SUM_LOCK_TIME` and `SUM_ERRORS` columns in statement summary tables are the aggregates of the `LOCK_TIME` and `ERRORS` columns in `events_statements_current` table.

- `MAX_CONTROLLED_MEMORY`

  Reports the maximum amount of controlled memory used by a statement during execution.

  This column was added in MySQL 8.0.31.

- `MAX_TOTAL_MEMORY`

  Reports the maximum amount of memory used by a statement during execution.

  This column was added in MySQL 8.0.31.

- `COUNT_SECONDARY`

The number of times a query was processed on the SECONDARY engine. For use with HeatWave Service and HeatWave, where the PRIMARY engine is InnoDB and the SECONDARY engine is HeatWave (RAPID). For MySQL Community Edition Server, MySQL Enterprise Edition Server (on-premise), and HeatWave Service without HeatWave, queries are always processed on the PRIMARY engine, which means the value is always 0 on these MySQL Servers. The COUNT_SECONDARY column was added in MySQL 8.0.29.

The events_statements_summary_by_digest table has these additional summary columns:

- FIRST_SEEN, LAST_SEEN

  Timestamps indicating when statements with the given digest value were first seen and most recently seen.

- QUANTILE_95: The 95th percentile of the statement latency, in picoseconds. This percentile is a high estimate, computed from the histogram data collected. In other words, for a given digest, 95% of the statements measured have a latency lower than QUANTILE_95.

  For access to the histogram data, use the tables described in Section 10.20.4, “Statement Histogram Summary Tables”.

- QUANTILE_99: Similar to QUANTILE_95, but for the 99th percentile.

- QUANTILE_999: Similar to QUANTILE_95, but for the 99.9th percentile.

The events_statements_summary_by_digest table contains the following columns. These are neither grouping nor summary columns; they support statement sampling:

- QUERY_SAMPLE_TEXT

  A sample SQL statement that produces the digest value in the row. This column enables applications to access, for a given digest value, a statement actually seen by the server that produces that digest. One use for this might be to run EXPLAIN on the statement to examine the execution plan for a representative statement associated with a frequently occurring digest.

  When the QUERY_SAMPLE_TEXT column is assigned a value, the QUERY_SAMPLE_SEEN and QUERY_SAMPLE_TIMER_WAIT columns are assigned values as well.

  The maximum space available for statement display is 1024 bytes by default. To change this value, set the performance_schema_max_sql_text_length system variable at server startup. (Changing this value affects columns in other Performance Schema tables as well. See Performance Schema Statement Digests and Sampling.)

  For information about statement sampling, see Performance Schema Statement Digests and Sampling.

- QUERY_SAMPLE_SEEN

  A timestamp indicating when the statement in the QUERY_SAMPLE_TEXT column was seen.

- QUERY_SAMPLE_TIMER_WAIT

  The wait time for the sample statement in the QUERY_SAMPLE_TEXT column.

The events_statements_summary_by_program table has these additional summary columns:

- COUNT_STATEMENTS, SUM_STATEMENTS_WAIT, MIN_STATEMENTS_WAIT, AVG_STATEMENTS_WAIT, MAX_STATEMENTS_WAIT

Statistics about nested statements invoked during stored program execution.

The `prepared_statements_instances` table has these additional summary columns:

- `COUNT_EXECUTE`, `SUM_TIMER_EXECUTE`, `MIN_TIMER_EXECUTE`, `AVG_TIMER_EXECUTE`, `MAX_TIMER_EXECUTE`

  Aggregated statistics for executions of the prepared statement.

The statement summary tables have these indexes:

- `events_transactions_summary_by_account_by_event_name`:
  - Primary key on (`USER`, `HOST`, `EVENT_NAME`)
- `events_statements_summary_by_digest`:
  - Primary key on (`SCHEMA_NAME`, `DIGEST`)
- `events_transactions_summary_by_host_by_event_name`:
  - Primary key on (`HOST`, `EVENT_NAME`)
- `events_statements_summary_by_program`:
  - Primary key on (`OBJECT_TYPE`, `OBJECT_SCHEMA`, `OBJECT_NAME`)
- `events_statements_summary_by_thread_by_event_name`:
  - Primary key on (`THREAD_ID`, `EVENT_NAME`)
- `events_transactions_summary_by_user_by_event_name`:
  - Primary key on (`USER`, `EVENT_NAME`)
- `events_statements_summary_global_by_event_name`:
  - Primary key on (`EVENT_NAME`)

`TRUNCATE TABLE` is permitted for statement summary tables. It has these effects:

- For `events_statements_summary_by_digest`, it removes the rows.
- For other summary tables not aggregated by account, host, or user, truncation resets the summary columns to zero rather than removing rows.
- For other summary tables aggregated by account, host, or user, truncation removes rows for accounts, hosts, or users with no connections, and resets the summary columns to zero for the remaining rows.

In addition, each statement summary table that is aggregated by account, host, user, or thread is implicitly truncated by truncation of the connection table on which it depends, or truncation of `events_statements_summary_global_by_event_name`. For details, see Section 10.8, “Performance Schema Connection Tables”.

In addition, truncating `events_statements_summary_by_digest` implicitly truncates `events_statements_histogram_by_digest`, and truncating `events_statements_summary_global_by_event_name` implicitly truncates `events_statements_histogram_global`.

Statement Digest Aggregation Rules

If the `statements_digest` consumer is enabled, aggregation into `events_statements_summary_by_digest` occurs as follows when a statement completes. Aggregation is based on the `DIGEST` value computed for the statement.

- If a `events_statements_summary_by_digest` row already exists with the digest value for the statement that just completed, statistics for the statement are aggregated to that row. The `LAST_SEEN` column is updated to the current time.
- If no row has the digest value for the statement that just completed, and the table is not full, a new row is created for the statement. The `FIRST_SEEN` and `LAST_SEEN` columns are initialized with the current time.
- If no row has the statement digest value for the statement that just completed, and the table is full, the statistics for the statement that just completed are added to a special "catch-all" row with `DIGEST` = `NULL`, which is created if necessary. If the row is created, the `FIRST_SEEN` and `LAST_SEEN` columns are initialized with the current time. Otherwise, the `LAST_SEEN` column is updated with the current time.

The row with `DIGEST` = `NULL` is maintained because Performance Schema tables have a maximum size due to memory constraints. The `DIGEST` = `NULL` row permits digests that do not match other rows to be counted even if the summary table is full, using a common "other" bucket. This row helps you estimate whether the digest summary is representative:

- A `DIGEST` = `NULL` row that has a `COUNT_STAR` value that represents 5% of all digests shows that the digest summary table is very representative; the other rows cover 95% of the statements seen.
- A `DIGEST` = `NULL` row that has a `COUNT_STAR` value that represents 50% of all digests shows that the digest summary table is not very representative; the other rows cover only half the statements seen. Most likely the DBA should increase the maximum table size so that more of the rows counted in the `DIGEST` = `NULL` row would be counted using more specific rows instead. By default, the table is autosized, but if this size is too small, set the `performance_schema_digests_size` system variable to a larger value at server startup.

Stored Program Instrumentation Behavior

For stored program types for which instrumentation is enabled in the `setup_objects` table, `events_statements_summary_by_program` maintains statistics for stored programs as follows:

- A row is added for an object when it is first used in the server.
- The row for an object is removed when the object is dropped.
- Statistics are aggregated in the row for an object as it executes.

See also Section 5.3, "Event Pre-Filtering".

10.20.4 Statement Histogram Summary Tables

The Performance Schema maintains statement event summary tables that contain information about minimum, maximum, and average statement latency (see Section 10.20.3, "Statement Summary Tables"). Those tables permit high-level assessment of system performance. To permit assessment at a more fine-grained level, the Performance Schema also collects histogram data for statement latencies. These histograms provide additional insight into latency distributions.

Section 10.6, "Performance Schema Statement Event Tables" describes the events on which statement summaries are based. See that discussion for information about the content of statement events, the

current and historical statement event tables, and how to control statement event collection, which is partially disabled by default.

Example statement histogram information:

```
mysql> SELECT *
       FROM performance_schema.events_statements_histogram_by_digest
       WHERE SCHEMA_NAME = 'mydb' AND DIGEST = 'bb3f69453119b2d7b3ae40673a9d4c7c'
       AND COUNT_BUCKET > 0 ORDER BY BUCKET_NUMBER\G
*************************** 1. row ***************************
           SCHEMA_NAME: mydb
                DIGEST: bb3f69453119b2d7b3ae40673a9d4c7c
         BUCKET_NUMBER: 42
      BUCKET_TIMER_LOW: 66069344
     BUCKET_TIMER_HIGH: 69183097
          COUNT_BUCKET: 1
COUNT_BUCKET_AND_LOWER: 1
       BUCKET_QUANTILE: 0.058824
*************************** 2. row ***************************
           SCHEMA_NAME: mydb
                DIGEST: bb3f69453119b2d7b3ae40673a9d4c7c
         BUCKET_NUMBER: 43
      BUCKET_TIMER_LOW: 69183097
     BUCKET_TIMER_HIGH: 72443596
          COUNT_BUCKET: 1
COUNT_BUCKET_AND_LOWER: 2
       BUCKET_QUANTILE: 0.117647
*************************** 3. row ***************************
           SCHEMA_NAME: mydb
                DIGEST: bb3f69453119b2d7b3ae40673a9d4c7c
         BUCKET_NUMBER: 44
      BUCKET_TIMER_LOW: 72443596
     BUCKET_TIMER_HIGH: 75857757
          COUNT_BUCKET: 2
COUNT_BUCKET_AND_LOWER: 4
       BUCKET_QUANTILE: 0.235294
*************************** 4. row ***************************
           SCHEMA_NAME: mydb
                DIGEST: bb3f69453119b2d7b3ae40673a9d4c7c
         BUCKET_NUMBER: 45
      BUCKET_TIMER_LOW: 75857757
     BUCKET_TIMER_HIGH: 79432823
          COUNT_BUCKET: 6
COUNT_BUCKET_AND_LOWER: 10
       BUCKET_QUANTILE: 0.625000
...
```

For example, in row 3, these values indicate that 23.52% of queries run in under 75.86 microseconds:

```
BUCKET_TIMER_HIGH: 75857757
  BUCKET_QUANTILE: 0.235294
```

In row 4, these values indicate that 62.50% of queries run in under 79.44 microseconds:

```
BUCKET_TIMER_HIGH: 79432823
  BUCKET_QUANTILE: 0.625000
```

Each statement histogram summary table has one or more grouping columns to indicate how the table aggregates events:

- `events_statements_histogram_by_digest` has `SCHEMA_NAME`, `DIGEST`, and `BUCKET_NUMBER` columns:
  - The `SCHEMA_NAME` and `DIGEST` columns identify a statement digest row in the `events_statements_summary_by_digest` table.

  - The `events_statements_histogram_by_digest` rows with the same `SCHEMA_NAME` and `DIGEST` values comprise the histogram for that schema/digest combination.
  - Within a given histogram, the `BUCKET_NUMBER` column indicates the bucket number.
- `events_statements_histogram_global` has a `BUCKET_NUMBER` column. This table summarizes latencies globally across schema name and digest values, using a single histogram. The `BUCKET_NUMBER` column indicates the bucket number within this global histogram.

A histogram consists of *N* buckets, where each row represents one bucket, with the bucket number indicated by the `BUCKET_NUMBER` column. Bucket numbers begin with 0.

Each statement histogram summary table has these summary columns containing aggregated values:

- `BUCKET_TIMER_LOW`, `BUCKET_TIMER_HIGH`

  A bucket counts statements that have a latency, in picoseconds, measured between `BUCKET_TIMER_LOW` and `BUCKET_TIMER_HIGH`:

  - The value of `BUCKET_TIMER_LOW` for the first bucket (`BUCKET_NUMBER` = 0) is 0.
  - The value of `BUCKET_TIMER_LOW` for a bucket (`BUCKET_NUMBER` = *k*) is the same as `BUCKET_TIMER_HIGH` for the previous bucket (`BUCKET_NUMBER` = *k*−1)
  - The last bucket is a catchall for statements that have a latency exceeding previous buckets in the histogram.
- `COUNT_BUCKET`

  The number of statements measured with a latency in the interval from `BUCKET_TIMER_LOW` up to but not including `BUCKET_TIMER_HIGH`.
- `COUNT_BUCKET_AND_LOWER`

  The number of statements measured with a latency in the interval from 0 up to but not including `BUCKET_TIMER_HIGH`.
- `BUCKET_QUANTILE`

  The proportion of statements that fall into this or a lower bucket. This proportion corresponds by definition to `COUNT_BUCKET_AND_LOWER / SUM(COUNT_BUCKET)` and is displayed as a convenience column.

The statement histogram summary tables have these indexes:

- `events_statements_histogram_by_digest`:
  - Unique index on (`SCHEMA_NAME`, `DIGEST`, `BUCKET_NUMBER`)
- `events_statements_histogram_global`:
  - Primary key on (`BUCKET_NUMBER`)

`TRUNCATE TABLE` is permitted for statement histogram summary tables. Truncation sets the `COUNT_BUCKET` and `COUNT_BUCKET_AND_LOWER` columns to 0.

In addition, truncating `events_statements_summary_by_digest` implicitly truncates `events_statements_histogram_by_digest`, and truncating

`events_statements_summary_global_by_event_name` implicitly truncates `events_statements_histogram_global`.

10.20.5 Transaction Summary Tables

The Performance Schema maintains tables for collecting current and recent transaction events, and aggregates that information in summary tables. Section 10.7, “Performance Schema Transaction Tables” describes the events on which transaction summaries are based. See that discussion for information about the content of transaction events, the current and historical transaction event tables, and how to control transaction event collection, which is disabled by default.

Example transaction event summary information:

```
mysql> SELECT *
       FROM performance_schema.events_transactions_summary_global_by_event_name
       LIMIT 1\G
*************************** 1. row ***************************
          EVENT_NAME: transaction
          COUNT_STAR: 5
      SUM_TIMER_WAIT: 19550092000
      MIN_TIMER_WAIT: 2954148000
      AVG_TIMER_WAIT: 3910018000
      MAX_TIMER_WAIT: 5486275000
    COUNT_READ_WRITE: 5
SUM_TIMER_READ_WRITE: 19550092000
MIN_TIMER_READ_WRITE: 2954148000
AVG_TIMER_READ_WRITE: 3910018000
MAX_TIMER_READ_WRITE: 5486275000
     COUNT_READ_ONLY: 0
 SUM_TIMER_READ_ONLY: 0
 MIN_TIMER_READ_ONLY: 0
 AVG_TIMER_READ_ONLY: 0
 MAX_TIMER_READ_ONLY: 0
```

Each transaction summary table has one or more grouping columns to indicate how the table aggregates events. Event names refer to names of event instruments in the `setup_instruments` table:

- `events_transactions_summary_by_account_by_event_name` has `USER`, `HOST`, and `EVENT_NAME` columns. Each row summarizes events for a given account (user and host combination) and event name.
- `events_transactions_summary_by_host_by_event_name` has `HOST` and `EVENT_NAME` columns. Each row summarizes events for a given host and event name.
- `events_transactions_summary_by_thread_by_event_name` has `THREAD_ID` and `EVENT_NAME` columns. Each row summarizes events for a given thread and event name.
- `events_transactions_summary_by_user_by_event_name` has `USER` and `EVENT_NAME` columns. Each row summarizes events for a given user and event name.
- `events_transactions_summary_global_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name.

Each transaction summary table has these summary columns containing aggregated values:

- `COUNT_STAR`, `SUM_TIMER_WAIT`, `MIN_TIMER_WAIT`, `AVG_TIMER_WAIT`, `MAX_TIMER_WAIT`

  These columns are analogous to the columns of the same names in the wait event summary tables (see Section 10.20.1, “Wait Event Summary Tables”), except that the transaction summary tables aggregate events from `events_transactions_current` rather than `events_waits_current`. These columns summarize read-write and read-only transactions.

- `COUNT_READ_WRITE`, `SUM_TIMER_READ_WRITE`, `MIN_TIMER_READ_WRITE`, `AVG_TIMER_READ_WRITE`, `MAX_TIMER_READ_WRITE`

  These are similar to the `COUNT_STAR` and *`xxx`*`_TIMER_WAIT` columns, but summarize read-write transactions only. The transaction access mode specifies whether transactions operate in read/write or read-only mode.

- `COUNT_READ_ONLY`, `SUM_TIMER_READ_ONLY`, `MIN_TIMER_READ_ONLY`, `AVG_TIMER_READ_ONLY`, `MAX_TIMER_READ_ONLY`

  These are similar to the `COUNT_STAR` and *`xxx`*`_TIMER_WAIT` columns, but summarize read-only transactions only. The transaction access mode specifies whether transactions operate in read/write or read-only mode.

The transaction summary tables have these indexes:

- `events_transactions_summary_by_account_by_event_name`:
  - Primary key on (`USER`, `HOST`, `EVENT_NAME`)
- `events_transactions_summary_by_host_by_event_name`:
  - Primary key on (`HOST`, `EVENT_NAME`)
- `events_transactions_summary_by_thread_by_event_name`:
  - Primary key on (`THREAD_ID`, `EVENT_NAME`)
- `events_transactions_summary_by_user_by_event_name`:
  - Primary key on (`USER`, `EVENT_NAME`)
- `events_transactions_summary_global_by_event_name`:
  - Primary key on (`EVENT_NAME`)

`TRUNCATE TABLE` is permitted for transaction summary tables. It has these effects:

- For summary tables not aggregated by account, host, or user, truncation resets the summary columns to zero rather than removing rows.
- For summary tables aggregated by account, host, or user, truncation removes rows for accounts, hosts, or users with no connections, and resets the summary columns to zero for the remaining rows.

In addition, each transaction summary table that is aggregated by account, host, user, or thread is implicitly truncated by truncation of the connection table on which it depends, or truncation of `events_transactions_summary_global_by_event_name`. For details, see Section 10.8, “Performance Schema Connection Tables”.

Transaction Aggregation Rules

Transaction event collection occurs without regard to isolation level, access mode, or autocommit mode.

Transaction event collection occurs for all non-aborted transactions initiated by the server, including empty transactions.

Read-write transactions are generally more resource intensive than read-only transactions, therefore transaction summary tables include separate aggregate columns for read-write and read-only transactions.

Resource requirements may also vary with transaction isolation level. However, presuming that only one isolation level would be used per server, aggregation by isolation level is not provided.

10.20.6 Object Wait Summary Table

The Performance Schema maintains the `objects_summary_global_by_type` table for aggregating object wait events.

Example object wait event summary information:

```
mysql> SELECT * FROM performance_schema.objects_summary_global_by_type\G
...
*************************** 3. row ***************************
   OBJECT_TYPE: TABLE
 OBJECT_SCHEMA: test
   OBJECT_NAME: t
    COUNT_STAR: 3
SUM_TIMER_WAIT: 263126976
MIN_TIMER_WAIT: 1522272
AVG_TIMER_WAIT: 87708678
MAX_TIMER_WAIT: 258428280
...
*************************** 10. row ***************************
   OBJECT_TYPE: TABLE
 OBJECT_SCHEMA: mysql
   OBJECT_NAME: user
    COUNT_STAR: 14
SUM_TIMER_WAIT: 365567592
MIN_TIMER_WAIT: 1141704
AVG_TIMER_WAIT: 26111769
MAX_TIMER_WAIT: 334783032
...
```

The `objects_summary_global_by_type` table has these grouping columns to indicate how the table aggregates events: `OBJECT_TYPE`, `OBJECT_SCHEMA`, and `OBJECT_NAME`. Each row summarizes events for the given object.

`objects_summary_global_by_type` has the same summary columns as the `events_waits_summary_by_xxx` tables. See Section 10.20.1, “Wait Event Summary Tables”.

The `objects_summary_global_by_type` table has these indexes:

- Primary key on (`OBJECT_TYPE`, `OBJECT_SCHEMA`, `OBJECT_NAME`)

`TRUNCATE TABLE` is permitted for the object summary table. It resets the summary columns to zero rather than removing rows.

10.20.7 File I/O Summary Tables

The Performance Schema maintains file I/O summary tables that aggregate information about I/O operations.

Example file I/O event summary information:

```
mysql> SELECT * FROM performance_schema.file_summary_by_event_name\G
...
*************************** 2. row ***************************
              EVENT_NAME: wait/io/file/sql/binlog
              COUNT_STAR: 31
          SUM_TIMER_WAIT: 8243784888
```

```
           MIN_TIMER_WAIT: 0
           AVG_TIMER_WAIT: 265928484
           MAX_TIMER_WAIT: 6490658832
...
mysql> SELECT * FROM performance_schema.file_summary_by_instance\G
...
*************************** 2. row ***************************
               FILE_NAME: /var/mysql/share/english/errmsg.sys
              EVENT_NAME: wait/io/file/sql/ERRMSG
              EVENT_NAME: wait/io/file/sql/ERRMSG
    OBJECT_INSTANCE_BEGIN: 4686193384
              COUNT_STAR: 5
           SUM_TIMER_WAIT: 13990154448
           MIN_TIMER_WAIT: 26349624
           AVG_TIMER_WAIT: 2798030607
           MAX_TIMER_WAIT: 8150662536
...
```

Each file I/O summary table has one or more grouping columns to indicate how the table aggregates events. Event names refer to names of event instruments in the `setup_instruments` table:

- `file_summary_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name.
- `file_summary_by_instance` has `FILE_NAME`, `EVENT_NAME`, and `OBJECT_INSTANCE_BEGIN` columns. Each row summarizes events for a given file and event name.

Each file I/O summary table has the following summary columns containing aggregated values. Some columns are more general and have values that are the same as the sum of the values of more fine-grained columns. In this way, aggregations at higher levels are available directly without the need for user-defined views that sum lower-level columns.

- `COUNT_STAR`, `SUM_TIMER_WAIT`, `MIN_TIMER_WAIT`, `AVG_TIMER_WAIT`, `MAX_TIMER_WAIT`

  These columns aggregate all I/O operations.

- `COUNT_READ`, `SUM_TIMER_READ`, `MIN_TIMER_READ`, `AVG_TIMER_READ`, `MAX_TIMER_READ`, `SUM_NUMBER_OF_BYTES_READ`

  These columns aggregate all read operations, including `FGETS`, `FGETC`, `FREAD`, and `READ`.

- `COUNT_WRITE`, `SUM_TIMER_WRITE`, `MIN_TIMER_WRITE`, `AVG_TIMER_WRITE`, `MAX_TIMER_WRITE`, `SUM_NUMBER_OF_BYTES_WRITE`

  These columns aggregate all write operations, including `FPUTS`, `FPUTC`, `FPRINTF`, `VFPRINTF`, `FWRITE`, and `PWRITE`.

- `COUNT_MISC`, `SUM_TIMER_MISC`, `MIN_TIMER_MISC`, `AVG_TIMER_MISC`, `MAX_TIMER_MISC`

  These columns aggregate all other I/O operations, including `CREATE`, `DELETE`, `OPEN`, `CLOSE`, `STREAM_OPEN`, `STREAM_CLOSE`, `SEEK`, `TELL`, `FLUSH`, `STAT`, `FSTAT`, `CHSIZE`, `RENAME`, and `SYNC`. There are no byte counts for these operations.

The file I/O summary tables have these indexes:

- `file_summary_by_event_name`:
  - Primary key on (`EVENT_NAME`)
- `file_summary_by_instance`:

- Primary key on (OBJECT_INSTANCE_BEGIN)
- Index on (FILE_NAME)
- Index on (EVENT_NAME)

TRUNCATE TABLE is permitted for file I/O summary tables. It resets the summary columns to zero rather than removing rows.

The MySQL server uses several techniques to avoid I/O operations by caching information read from files, so it is possible that statements you might expect to result in I/O events do not do so. You may be able to ensure that I/O does occur by flushing caches or restarting the server to reset its state.

10.20.8 Table I/O and Lock Wait Summary Tables

The following sections describe the table I/O and lock wait summary tables:

- table_io_waits_summary_by_index_usage: Table I/O waits per index
- table_io_waits_summary_by_table: Table I/O waits per table
- table_lock_waits_summary_by_table: Table lock waits per table

10.20.8.1 The table_io_waits_summary_by_table Table

The table_io_waits_summary_by_table table aggregates all table I/O wait events, as generated by the wait/io/table/sql/handler instrument. The grouping is by table.

The table_io_waits_summary_by_table table has these grouping columns to indicate how the table aggregates events: OBJECT_TYPE, OBJECT_SCHEMA, and OBJECT_NAME. These columns have the same meaning as in the events_waits_current table. They identify the table to which the row applies.

table_io_waits_summary_by_table has the following summary columns containing aggregated values. As indicated in the column descriptions, some columns are more general and have values that are the same as the sum of the values of more fine-grained columns. For example, columns that aggregate all writes hold the sum of the corresponding columns that aggregate inserts, updates, and deletes. In this way, aggregations at higher levels are available directly without the need for user-defined views that sum lower-level columns.

- COUNT_STAR, SUM_TIMER_WAIT, MIN_TIMER_WAIT, AVG_TIMER_WAIT, MAX_TIMER_WAIT

  These columns aggregate all I/O operations. They are the same as the sum of the corresponding *xxx*_READ and *xxx*_WRITE columns.

- COUNT_READ, SUM_TIMER_READ, MIN_TIMER_READ, AVG_TIMER_READ, MAX_TIMER_READ

  These columns aggregate all read operations. They are the same as the sum of the corresponding *xxx*_FETCH columns.

- COUNT_WRITE, SUM_TIMER_WRITE, MIN_TIMER_WRITE, AVG_TIMER_WRITE, MAX_TIMER_WRITE

  These columns aggregate all write operations. They are the same as the sum of the corresponding *xxx*_INSERT, *xxx*_UPDATE, and *xxx*_DELETE columns.

- COUNT_FETCH, SUM_TIMER_FETCH, MIN_TIMER_FETCH, AVG_TIMER_FETCH, MAX_TIMER_FETCH

  These columns aggregate all fetch operations.

- `COUNT_INSERT`, `SUM_TIMER_INSERT`, `MIN_TIMER_INSERT`, `AVG_TIMER_INSERT`, `MAX_TIMER_INSERT`

  These columns aggregate all insert operations.

- `COUNT_UPDATE`, `SUM_TIMER_UPDATE`, `MIN_TIMER_UPDATE`, `AVG_TIMER_UPDATE`, `MAX_TIMER_UPDATE`

  These columns aggregate all update operations.

- `COUNT_DELETE`, `SUM_TIMER_DELETE`, `MIN_TIMER_DELETE`, `AVG_TIMER_DELETE`, `MAX_TIMER_DELETE`

  These columns aggregate all delete operations.

The `table_io_waits_summary_by_table` table has these indexes:

- Unique index on (`OBJECT_TYPE`, `OBJECT_SCHEMA`, `OBJECT_NAME`)

`TRUNCATE TABLE` is permitted for table I/O summary tables. It resets the summary columns to zero rather than removing rows. Truncating this table also truncates the `table_io_waits_summary_by_index_usage` table.

10.20.8.2 The table_io_waits_summary_by_index_usage Table

The `table_io_waits_summary_by_index_usage` table aggregates all table index I/O wait events, as generated by the `wait/io/table/sql/handler` instrument. The grouping is by table index.

The columns of `table_io_waits_summary_by_index_usage` are nearly identical to `table_io_waits_summary_by_table`. The only difference is the additional group column, `INDEX_NAME`, which corresponds to the name of the index that was used when the table I/O wait event was recorded:

- A value of `PRIMARY` indicates that table I/O used the primary index.
- A value of `NULL` means that table I/O used no index.
- Inserts are counted against `INDEX_NAME = NULL`.

The `table_io_waits_summary_by_index_usage` table has these indexes:

- Unique index on (`OBJECT_TYPE`, `OBJECT_SCHEMA`, `OBJECT_NAME`, `INDEX_NAME`)

`TRUNCATE TABLE` is permitted for table I/O summary tables. It resets the summary columns to zero rather than removing rows. This table is also truncated by truncation of the `table_io_waits_summary_by_table` table. A DDL operation that changes the index structure of a table may cause the per-index statistics to be reset.

10.20.8.3 The table_lock_waits_summary_by_table Table

The `table_lock_waits_summary_by_table` table aggregates all table lock wait events, as generated by the `wait/lock/table/sql/handler` instrument. The grouping is by table.

This table contains information about internal and external locks:

- An internal lock corresponds to a lock in the SQL layer. This is currently implemented by a call to `thr_lock()`. In event rows, these locks are distinguished by the `OPERATION` column, which has one of these values:

```
read normal
read with shared locks
read high priority
read no insert
write allow write
write concurrent insert
write delayed
write low priority
write normal
```

- An external lock corresponds to a lock in the storage engine layer. This is currently implemented by a call to `handler::external_lock()`. In event rows, these locks are distinguished by the `OPERATION` column, which has one of these values:

```
read external
write external
```

The `table_lock_waits_summary_by_table` table has these grouping columns to indicate how the table aggregates events: `OBJECT_TYPE`, `OBJECT_SCHEMA`, and `OBJECT_NAME`. These columns have the same meaning as in the `events_waits_current` table. They identify the table to which the row applies.

`table_lock_waits_summary_by_table` has the following summary columns containing aggregated values. As indicated in the column descriptions, some columns are more general and have values that are the same as the sum of the values of more fine-grained columns. For example, columns that aggregate all locks hold the sum of the corresponding columns that aggregate read and write locks. In this way, aggregations at higher levels are available directly without the need for user-defined views that sum lower-level columns.

- `COUNT_STAR`, `SUM_TIMER_WAIT`, `MIN_TIMER_WAIT`, `AVG_TIMER_WAIT`, `MAX_TIMER_WAIT`

  These columns aggregate all lock operations. They are the same as the sum of the corresponding *`xxx`*`_READ` and *`xxx`*`_WRITE` columns.

- `COUNT_READ`, `SUM_TIMER_READ`, `MIN_TIMER_READ`, `AVG_TIMER_READ`, `MAX_TIMER_READ`

  These columns aggregate all read-lock operations. They are the same as the sum of the corresponding *`xxx`*`_READ_NORMAL`, *`xxx`*`_READ_WITH_SHARED_LOCKS`, *`xxx`*`_READ_HIGH_PRIORITY`, and *`xxx`*`_READ_NO_INSERT` columns.

- `COUNT_WRITE`, `SUM_TIMER_WRITE`, `MIN_TIMER_WRITE`, `AVG_TIMER_WRITE`, `MAX_TIMER_WRITE`

  These columns aggregate all write-lock operations. They are the same as the sum of the corresponding *`xxx`*`_WRITE_ALLOW_WRITE`, *`xxx`*`_WRITE_CONCURRENT_INSERT`, *`xxx`*`_WRITE_LOW_PRIORITY`, and *`xxx`*`_WRITE_NORMAL` columns.

- `COUNT_READ_NORMAL`, `SUM_TIMER_READ_NORMAL`, `MIN_TIMER_READ_NORMAL`, `AVG_TIMER_READ_NORMAL`, `MAX_TIMER_READ_NORMAL`

  These columns aggregate internal read locks.

- `COUNT_READ_WITH_SHARED_LOCKS`, `SUM_TIMER_READ_WITH_SHARED_LOCKS`, `MIN_TIMER_READ_WITH_SHARED_LOCKS`, `AVG_TIMER_READ_WITH_SHARED_LOCKS`, `MAX_TIMER_READ_WITH_SHARED_LOCKS`

  These columns aggregate internal read locks.

- `COUNT_READ_HIGH_PRIORITY`, `SUM_TIMER_READ_HIGH_PRIORITY`, `MIN_TIMER_READ_HIGH_PRIORITY`, `AVG_TIMER_READ_HIGH_PRIORITY`, `MAX_TIMER_READ_HIGH_PRIORITY`

These columns aggregate internal read locks.

- `COUNT_READ_NO_INSERT`, `SUM_TIMER_READ_NO_INSERT`, `MIN_TIMER_READ_NO_INSERT`, `AVG_TIMER_READ_NO_INSERT`, `MAX_TIMER_READ_NO_INSERT`

  These columns aggregate internal read locks.

- `COUNT_READ_EXTERNAL`, `SUM_TIMER_READ_EXTERNAL`, `MIN_TIMER_READ_EXTERNAL`, `AVG_TIMER_READ_EXTERNAL`, `MAX_TIMER_READ_EXTERNAL`

  These columns aggregate external read locks.

- `COUNT_WRITE_ALLOW_WRITE`, `SUM_TIMER_WRITE_ALLOW_WRITE`, `MIN_TIMER_WRITE_ALLOW_WRITE`, `AVG_TIMER_WRITE_ALLOW_WRITE`, `MAX_TIMER_WRITE_ALLOW_WRITE`

  These columns aggregate internal write locks.

- `COUNT_WRITE_CONCURRENT_INSERT`, `SUM_TIMER_WRITE_CONCURRENT_INSERT`, `MIN_TIMER_WRITE_CONCURRENT_INSERT`, `AVG_TIMER_WRITE_CONCURRENT_INSERT`, `MAX_TIMER_WRITE_CONCURRENT_INSERT`

  These columns aggregate internal write locks.

- `COUNT_WRITE_LOW_PRIORITY`, `SUM_TIMER_WRITE_LOW_PRIORITY`, `MIN_TIMER_WRITE_LOW_PRIORITY`, `AVG_TIMER_WRITE_LOW_PRIORITY`, `MAX_TIMER_WRITE_LOW_PRIORITY`

  These columns aggregate internal write locks.

- `COUNT_WRITE_NORMAL`, `SUM_TIMER_WRITE_NORMAL`, `MIN_TIMER_WRITE_NORMAL`, `AVG_TIMER_WRITE_NORMAL`, `MAX_TIMER_WRITE_NORMAL`

  These columns aggregate internal write locks.

- `COUNT_WRITE_EXTERNAL`, `SUM_TIMER_WRITE_EXTERNAL`, `MIN_TIMER_WRITE_EXTERNAL`, `AVG_TIMER_WRITE_EXTERNAL`, `MAX_TIMER_WRITE_EXTERNAL`

  These columns aggregate external write locks.

The `table_lock_waits_summary_by_table` table has these indexes:

- Unique index on (`OBJECT_TYPE`, `OBJECT_SCHEMA`, `OBJECT_NAME`)

`TRUNCATE TABLE` is permitted for table lock summary tables. It resets the summary columns to zero rather than removing rows.

10.20.9 Socket Summary Tables

These socket summary tables aggregate timer and byte count information for socket operations:

- `socket_summary_by_event_name`: Aggregate timer and byte count statistics generated by the `wait/io/socket/*` instruments for all socket I/O operations, per socket instrument.
- `socket_summary_by_instance`: Aggregate timer and byte count statistics generated by the `wait/io/socket/*` instruments for all socket I/O operations, per socket instance. When a connection terminates, the row in `socket_summary_by_instance` corresponding to it is deleted.

The socket summary tables do not aggregate waits generated by `idle` events while sockets are waiting for the next request from the client. For `idle` event aggregations, use the wait-event summary tables; see Section 10.20.1, “Wait Event Summary Tables”.

Each socket summary table has one or more grouping columns to indicate how the table aggregates events. Event names refer to names of event instruments in the `setup_instruments` table:

- `socket_summary_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name.
- `socket_summary_by_instance` has an `OBJECT_INSTANCE_BEGIN` column. Each row summarizes events for a given object.

Each socket summary table has these summary columns containing aggregated values:

- `COUNT_STAR`, `SUM_TIMER_WAIT`, `MIN_TIMER_WAIT`, `AVG_TIMER_WAIT`, `MAX_TIMER_WAIT`

  These columns aggregate all operations.

- `COUNT_READ`, `SUM_TIMER_READ`, `MIN_TIMER_READ`, `AVG_TIMER_READ`, `MAX_TIMER_READ`, `SUM_NUMBER_OF_BYTES_READ`

  These columns aggregate all receive operations (`RECV`, `RECVFROM`, and `RECVMSG`).

- `COUNT_WRITE`, `SUM_TIMER_WRITE`, `MIN_TIMER_WRITE`, `AVG_TIMER_WRITE`, `MAX_TIMER_WRITE`, `SUM_NUMBER_OF_BYTES_WRITE`

  These columns aggregate all send operations (`SEND`, `SENDTO`, and `SENDMSG`).

- `COUNT_MISC`, `SUM_TIMER_MISC`, `MIN_TIMER_MISC`, `AVG_TIMER_MISC`, `MAX_TIMER_MISC`

  These columns aggregate all other socket operations, such as `CONNECT`, `LISTEN`, `ACCEPT`, `CLOSE`, and `SHUTDOWN`. There are no byte counts for these operations.

The `socket_summary_by_instance` table also has an `EVENT_NAME` column that indicates the class of the socket: `client_connection`, `server_tcpip_socket`, `server_unix_socket`. This column can be grouped on to isolate, for example, client activity from that of the server listening sockets.

The socket summary tables have these indexes:

- `socket_summary_by_event_name`:
  - Primary key on (`EVENT_NAME`)
- `socket_summary_by_instance`:
  - Primary key on (`OBJECT_INSTANCE_BEGIN`)
  - Index on (`EVENT_NAME`)

`TRUNCATE TABLE` is permitted for socket summary tables. Except for `events_statements_summary_by_digest`, it resets the summary columns to zero rather than removing rows.

10.20.10 Memory Summary Tables

The Performance Schema instruments memory usage and aggregates memory usage statistics, detailed by these factors:

- Type of memory used (various caches, internal buffers, and so forth)
- Thread, account, user, host indirectly performing the memory operation

The Performance Schema instruments the following aspects of memory use

- Memory sizes used
- Operation counts
- Low and high water marks

Memory sizes help to understand or tune the memory consumption of the server.

Operation counts help to understand or tune the overall pressure the server is putting on the memory allocator, which has an impact on performance. Allocating a single byte one million times is not the same as allocating one million bytes a single time; tracking both sizes and counts can expose the difference.

Low and high water marks are critical to detect workload spikes, overall workload stability, and possible memory leaks.

Memory summary tables do not contain timing information because memory events are not timed.

For information about collecting memory usage data, see Memory Instrumentation Behavior.

Example memory event summary information:

```
mysql> SELECT *
       FROM performance_schema.memory_summary_global_by_event_name
       WHERE EVENT_NAME = 'memory/sql/TABLE'\G
*************************** 1. row ***************************
                 EVENT_NAME: memory/sql/TABLE
                COUNT_ALLOC: 1381
                 COUNT_FREE: 924
   SUM_NUMBER_OF_BYTES_ALLOC: 2059873
    SUM_NUMBER_OF_BYTES_FREE: 1407432
              LOW_COUNT_USED: 0
          CURRENT_COUNT_USED: 457
             HIGH_COUNT_USED: 461
    LOW_NUMBER_OF_BYTES_USED: 0
CURRENT_NUMBER_OF_BYTES_USED: 652441
   HIGH_NUMBER_OF_BYTES_USED: 669269
```

Each memory summary table has one or more grouping columns to indicate how the table aggregates events. Event names refer to names of event instruments in the `setup_instruments` table:

- `memory_summary_by_account_by_event_name` has `USER`, `HOST`, and `EVENT_NAME` columns. Each row summarizes events for a given account (user and host combination) and event name.
- `memory_summary_by_host_by_event_name` has `HOST` and `EVENT_NAME` columns. Each row summarizes events for a given host and event name.
- `memory_summary_by_thread_by_event_name` has `THREAD_ID` and `EVENT_NAME` columns. Each row summarizes events for a given thread and event name.
- `memory_summary_by_user_by_event_name` has `USER` and `EVENT_NAME` columns. Each row summarizes events for a given user and event name.
- `memory_summary_global_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name.

Each memory summary table has these summary columns containing aggregated values:

- `COUNT_ALLOC`, `COUNT_FREE`

  The aggregated numbers of calls to memory-allocation and memory-free functions.

- `SUM_NUMBER_OF_BYTES_ALLOC`, `SUM_NUMBER_OF_BYTES_FREE`

  The aggregated sizes of allocated and freed memory blocks.

- `CURRENT_COUNT_USED`

  The aggregated number of currently allocated blocks that have not been freed yet. This is a convenience column, equal to `COUNT_ALLOC` − `COUNT_FREE`.

- `CURRENT_NUMBER_OF_BYTES_USED`

  The aggregated size of currently allocated memory blocks that have not been freed yet. This is a convenience column, equal to `SUM_NUMBER_OF_BYTES_ALLOC` − `SUM_NUMBER_OF_BYTES_FREE`.

- `LOW_COUNT_USED`, `HIGH_COUNT_USED`

  The low and high water marks corresponding to the `CURRENT_COUNT_USED` column.

- `LOW_NUMBER_OF_BYTES_USED`, `HIGH_NUMBER_OF_BYTES_USED`

  The low and high water marks corresponding to the `CURRENT_NUMBER_OF_BYTES_USED` column.

The memory summary tables have these indexes:

- `memory_summary_by_account_by_event_name`:
  - Primary key on (`USER`, `HOST`, `EVENT_NAME`)
- `memory_summary_by_host_by_event_name`:
  - Primary key on (`HOST`, `EVENT_NAME`)
- `memory_summary_by_thread_by_event_name`:
  - Primary key on (`THREAD_ID`, `EVENT_NAME`)
- `memory_summary_by_user_by_event_name`:
  - Primary key on (`USER`, `EVENT_NAME`)
- `memory_summary_global_by_event_name`:
  - Primary key on (`EVENT_NAME`)

`TRUNCATE TABLE` is permitted for memory summary tables. It has these effects:

- In general, truncation resets the baseline for statistics, but does not change the server state. That is, truncating a memory table does not free memory.
- `COUNT_ALLOC` and `COUNT_FREE` are reset to a new baseline, by reducing each counter by the same value.
- Likewise, `SUM_NUMBER_OF_BYTES_ALLOC` and `SUM_NUMBER_OF_BYTES_FREE` are reset to a new baseline.
- `LOW_COUNT_USED` and `HIGH_COUNT_USED` are reset to `CURRENT_COUNT_USED`.

- `LOW_NUMBER_OF_BYTES_USED` and `HIGH_NUMBER_OF_BYTES_USED` are reset to `CURRENT_NUMBER_OF_BYTES_USED`.

In addition, each memory summary table that is aggregated by account, host, user, or thread is implicitly truncated by truncation of the connection table on which it depends, or truncation of `memory_summary_global_by_event_name`. For details, see Section 10.8, "Performance Schema Connection Tables".

Memory Instrumentation Behavior

Memory instruments are listed in the `setup_instruments` table and have names of the form `memory/code_area/instrument_name`. Memory instrumentation is enabled by default.

Instruments named with the prefix `memory/performance_schema/` expose how much memory is allocated for internal buffers in the Performance Schema itself. The `memory/performance_schema/` instruments are built in, always enabled, and cannot be disabled at startup or runtime. Built-in memory instruments are displayed only in the `memory_summary_global_by_event_name` table.

To control memory instrumentation state at server startup, use lines like these in your `my.cnf` file:

- Enable:

```
[mysqld]
performance-schema-instrument='memory/%=ON'
```

- Disable:

```
[mysqld]
performance-schema-instrument='memory/%=OFF'
```

To control memory instrumentation state at runtime, update the `ENABLED` column of the relevant instruments in the `setup_instruments` table:

- Enable:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'YES'
WHERE NAME LIKE 'memory/%';
```

- Disable:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'NO'
WHERE NAME LIKE 'memory/%';
```

For memory instruments, the `TIMED` column in `setup_instruments` is ignored because memory operations are not timed.

When a thread in the server executes a memory allocation that has been instrumented, these rules apply:

- If the thread is not instrumented or the memory instrument is not enabled, the memory block allocated is not instrumented.
- Otherwise (that is, both the thread and the instrument are enabled), the memory block allocated is instrumented.

For deallocation, these rules apply:

- If a memory allocation operation was instrumented, the corresponding free operation is instrumented, regardless of the current instrument or thread enabled status.

- If a memory allocation operation was not instrumented, the corresponding free operation is not instrumented, regardless of the current instrument or thread enabled status.

For the per-thread statistics, the following rules apply.

When an instrumented memory block of size *N* is allocated, the Performance Schema makes these updates to memory summary table columns:

- `COUNT_ALLOC`: Increased by 1
- `CURRENT_COUNT_USED`: Increased by 1
- `HIGH_COUNT_USED`: Increased if `CURRENT_COUNT_USED` is a new maximum
- `SUM_NUMBER_OF_BYTES_ALLOC`: Increased by *N*
- `CURRENT_NUMBER_OF_BYTES_USED`: Increased by *N*
- `HIGH_NUMBER_OF_BYTES_USED`: Increased if `CURRENT_NUMBER_OF_BYTES_USED` is a new maximum

When an instrumented memory block is deallocated, the Performance Schema makes these updates to memory summary table columns:

- `COUNT_FREE`: Increased by 1
- `CURRENT_COUNT_USED`: Decreased by 1
- `LOW_COUNT_USED`: Decreased if `CURRENT_COUNT_USED` is a new minimum
- `SUM_NUMBER_OF_BYTES_FREE`: Increased by *N*
- `CURRENT_NUMBER_OF_BYTES_USED`: Decreased by *N*
- `LOW_NUMBER_OF_BYTES_USED`: Decreased if `CURRENT_NUMBER_OF_BYTES_USED` is a new minimum

For higher-level aggregates (global, by account, by user, by host), the same rules apply as expected for low and high water marks.

- `LOW_COUNT_USED` and `LOW_NUMBER_OF_BYTES_USED` are lower estimates. The value reported by the Performance Schema is guaranteed to be less than or equal to the lowest count or size of memory effectively used at runtime.
- `HIGH_COUNT_USED` and `HIGH_NUMBER_OF_BYTES_USED` are higher estimates. The value reported by the Performance Schema is guaranteed to be greater than or equal to the highest count or size of memory effectively used at runtime.

For lower estimates in summary tables other than `memory_summary_global_by_event_name`, it is possible for values to go negative if memory ownership is transferred between threads.

Here is an example of estimate computation; but note that estimate implementation is subject to change:

Thread 1 uses memory in the range from 1MB to 2MB during execution, as reported by the `LOW_NUMBER_OF_BYTES_USED` and `HIGH_NUMBER_OF_BYTES_USED` columns of the `memory_summary_by_thread_by_event_name` table.

Thread 2 uses memory in the range from 10MB to 12MB during execution, as reported likewise.

When these two threads belong to the same user account, the per-account summary estimates that this account used memory in the range from 11MB to 14MB. That is, the `LOW_NUMBER_OF_BYTES_USED` for the higher level aggregate is the sum of each `LOW_NUMBER_OF_BYTES_USED` (assuming the worst

case). Likewise, the HIGH_NUMBER_OF_BYTES_USED for the higher level aggregate is the sum of each HIGH_NUMBER_OF_BYTES_USED (assuming the worst case).

11MB is a lower estimate that can occur only if both threads hit the low usage mark at the same time.

14MB is a higher estimate that can occur only if both threads hit the high usage mark at the same time.

The real memory usage for this account could have been in the range from 11.5MB to 13.5MB.

For capacity planning, reporting the worst case is actually the desired behavior, as it shows what can potentially happen when sessions are uncorrelated, which is typically the case.

10.20.11 Error Summary Tables

The Performance Schema maintains summary tables for aggregating statistical information about server errors (and warnings). For a list of server errors, see Server Error Message Reference.

Collection of error information is controlled by the error instrument, which is enabled by default. Timing information is not collected.

Each error summary table has three columns that identify the error:

- ERROR_NUMBER is the numeric error value. The value is unique.
- ERROR_NAME is the symbolic error name corresponding to the ERROR_NUMBER value. The value is unique.
- SQLSTATE is the SQLSTATE value corresponding to the ERROR_NUMBER value. The value is not necessarily unique.

For example, if ERROR_NUMBER is 1050, ERROR_NAME is ER_TABLE_EXISTS_ERROR and SQLSTATE is 42S01.

Example error event summary information:

```
mysql> SELECT *
       FROM performance_schema.events_errors_summary_global_by_error
       WHERE SUM_ERROR_RAISED <> 0\G
*************************** 1. row ***************************
     ERROR_NUMBER: 1064
       ERROR_NAME: ER_PARSE_ERROR
        SQL_STATE: 42000
 SUM_ERROR_RAISED: 1
SUM_ERROR_HANDLED: 0
       FIRST_SEEN: 2016-06-28 07:34:02
        LAST_SEEN: 2016-06-28 07:34:02
*************************** 2. row ***************************
     ERROR_NUMBER: 1146
       ERROR_NAME: ER_NO_SUCH_TABLE
        SQL_STATE: 42S02
 SUM_ERROR_RAISED: 2
SUM_ERROR_HANDLED: 0
       FIRST_SEEN: 2016-06-28 07:34:05
        LAST_SEEN: 2016-06-28 07:36:18
*************************** 3. row ***************************
     ERROR_NUMBER: 1317
       ERROR_NAME: ER_QUERY_INTERRUPTED
        SQL_STATE: 70100
 SUM_ERROR_RAISED: 1
SUM_ERROR_HANDLED: 0
       FIRST_SEEN: 2016-06-28 11:01:49
        LAST_SEEN: 2016-06-28 11:01:49
```

Each error summary table has one or more grouping columns to indicate how the table aggregates errors:

- `events_errors_summary_by_account_by_error` has `USER`, `HOST`, and `ERROR_NUMBER` columns. Each row summarizes events for a given account (user and host combination) and error.
- `events_errors_summary_by_host_by_error` has `HOST` and `ERROR_NUMBER` columns. Each row summarizes events for a given host and error.
- `events_errors_summary_by_thread_by_error` has `THREAD_ID` and `ERROR_NUMBER` columns. Each row summarizes events for a given thread and error.
- `events_errors_summary_by_user_by_error` has `USER` and `ERROR_NUMBER` columns. Each row summarizes events for a given user and error.
- `events_errors_summary_global_by_error` has an `ERROR_NUMBER` column. Each row summarizes events for a given error.

Each error summary table has these summary columns containing aggregated values:

- `SUM_ERROR_RAISED`

  This column aggregates the number of times the error occurred.

- `SUM_ERROR_HANDLED`

  This column aggregates the number of times the error was handled by an SQL exception handler.

- `FIRST_SEEN`, `LAST_SEEN`

  Timestamp indicating when the error was first seen and most recently seen.

A `NULL` row in each error summary table is used to aggregate statistics for all errors that lie out of range of the instrumented errors. For example, if MySQL Server errors lie in the range from *`M`* to *`N`* and an error is raised with number *`Q`* not in that range, the error is aggregated in the `NULL` row. The `NULL` row is the row with `ERROR_NUMBER=0`, `ERROR_NAME=NULL`, and `SQLSTATE=NULL`.

The error summary tables have these indexes:

- `events_errors_summary_by_account_by_error`:
  - Primary key on (`USER`, `HOST`, `ERROR_NUMBER`)
- `events_errors_summary_by_host_by_error`:
  - Primary key on (`HOST`, `ERROR_NUMBER`)
- `events_errors_summary_by_thread_by_error`:
  - Primary key on (`THREAD_ID`, `ERROR_NUMBER`)
- `events_errors_summary_by_user_by_error`:
  - Primary key on (`USER`, `ERROR_NUMBER`)
- `events_errors_summary_global_by_error`:
  - Primary key on (`ERROR_NUMBER`)

`TRUNCATE TABLE` is permitted for error summary tables. It has these effects:

- For summary tables not aggregated by account, host, or user, truncation resets the summary columns to zero or `NULL` rather than removing rows.
- For summary tables aggregated by account, host, or user, truncation removes rows for accounts, hosts, or users with no connections, and resets the summary columns to zero or `NULL` for the remaining rows.

In addition, each error summary table that is aggregated by account, host, user, or thread is implicitly truncated by truncation of the connection table on which it depends, or truncation of `events_errors_summary_global_by_error`. For details, see Section 10.8, “Performance Schema Connection Tables”.

10.20.12 Status Variable Summary Tables

The Performance Schema makes status variable information available in the tables described in Section 10.15, “Performance Schema Status Variable Tables”. It also makes aggregated status variable information available in summary tables, described here. Each status variable summary table has one or more grouping columns to indicate how the table aggregates status values:

- `status_by_account` has `USER`, `HOST`, and `VARIABLE_NAME` columns to summarize status variables by account.
- `status_by_host` has `HOST` and `VARIABLE_NAME` columns to summarize status variables by the host from which clients connected.
- `status_by_user` has `USER` and `VARIABLE_NAME` columns to summarize status variables by client user name.

Each status variable summary table has this summary column containing aggregated values:

- `VARIABLE_VALUE`

  The aggregated status variable value for active and terminated sessions.

The status variable summary tables have these indexes:

- `status_by_account`:
  - Primary key on (`USER`, `HOST`, `VARIABLE_NAME`)
- `status_by_host`:
  - Primary key on (`HOST`, `VARIABLE_NAME`)
- `status_by_user`:
  - Primary key on (`USER`, `VARIABLE_NAME`)

The meaning of “account” in these tables is similar to its meaning in the MySQL grant tables in the `mysql` system database, in the sense that the term refers to a combination of user and host values. They differ in that, for grant tables, the host part of an account can be a pattern, whereas for Performance Schema tables, the host value is always a specific nonpattern host name.

Account status is collected when sessions terminate. The session status counters are added to the global status counters and the corresponding account status counters. If account statistics are not collected, the session status is added to host and user status, if host and user status are collected.

Account, host, and user statistics are not collected if the `performance_schema_accounts_size`, `performance_schema_hosts_size`, and `performance_schema_users_size` system variables, respectively, are set to 0.

The Performance Schema supports `TRUNCATE TABLE` for status variable summary tables as follows; in all cases, status for active sessions is unaffected:

- `status_by_account`: Aggregates account status from terminated sessions to user and host status, then resets account status.
- `status_by_host`: Resets aggregated host status from terminated sessions.
- `status_by_user`: Resets aggregated user status from terminated sessions.

`FLUSH STATUS` adds the session status from all active sessions to the global status variables, resets the status of all active sessions, and resets account, host, and user status values aggregated from disconnected sessions.

10.21 Performance Schema Miscellaneous Tables

The following sections describe tables that do not fall into the table categories discussed in the preceding sections:

- `component_scheduler_tasks`: The current status of each scheduled task.
- `error_log`: The most recent events written to the error log.
- `host_cache`: Information from the internal host cache.
- `innodb_redo_log_files`: Information about InnoDB redo log files.
- `log_status`: Information about server logs for backup purposes.
- `performance_timers`: Which event timers are available.
- `processlist`: Information about server processes.
- `threads`: Information about server threads.
- `tls_channel_status`: TLS context properties for connection interfaces.
- `user_defined_functions`: Loadable functions registered by a component, plugin, or `CREATE FUNCTION` statement.

10.21.1 The component_scheduler_tasks Table

The `component_scheduler_tasks` table contains a row for each scheduled task. Each row contains information about the ongoing progress of a task that applications, components, and plugins can implement, optionally, using the `scheduler` component (see Scheduler Component). For example, the `audit_log` server plugin utilizes the `scheduler` component to run a regular, recurring flush of its memory cache:

```
mysql> select * from performance_schema.component_scheduler_tasks\G
*************************** 1. row ***************************
           NAME: plugin_audit_log_flush_scheduler
         STATUS: WAITING
        COMMENT: Registered by the audit log plugin. Does a periodic refresh of the audit log
                 in-memory rules cache by calling audit_log_flush
INTERVAL_SECONDS: 100
       TIMES_RUN: 5
    TIMES_FAILED: 0
1 row in set (0.02 sec)
```

The `component_scheduler_tasks` table has the following columns:

- `NAME`

  The name supplied during the registration.

- `STATUS`

  The values are:

  - `RUNNING` if the task is active and being executed.
  - `WAITING` if the task is idle and waiting for the background thread to pick it up or waiting for the next time it needs to be run to arrive.

- `COMMENT`

  A compile-time comment provided by an application, component, or plugin. In the previous example, MySQL Enterprise Audit provides the comment using a server plugin named `audit_log`.

- `INTERVAL_SECONDS`

  The time in seconds to run a task, which an application, component, or plugin provides. MySQL Enterprise Audit enables you to specify this value using the `audit_log_flush_interval_seconds` system variable.

- `TIMES_RUN`

  A counter that increments by one every time the task runs successfully. It wraps around.

- `TIMES_FAILED`

  A counter that increments by one every time the execution of the task fails. It wraps around.

10.21.2 The error_log Table

Of the logs the MySQL server maintains, one is the error log to which it writes diagnostic messages (see The Error Log). Typically, the server writes diagnostics to a file on the server host or to a system log service. As of MySQL 8.0.22, depending on error log configuration, the server can also write the most recent error events to the Performance Schema `error_log` table. Granting the `SELECT` privilege for the `error_log` table thus gives clients and applications access to error log contents using SQL queries, enabling DBAs to provide access to the log without the need to permit direct file system access on the server host.

The `error_log` table supports focused queries based on its more structured columns. It also includes the full text of error messages to support more free-form analysis.

The table implementation uses a fixed-size, in-memory ring buffer, with old events automatically discarded as necessary to make room for new ones.

Example `error_log` contents:

```
mysql> SELECT * FROM performance_schema.error_log\G
*************************** 1. row ***************************
    LOGGED: 2020-08-06 09:25:00.338624
 THREAD_ID: 0
      PRIO: System
ERROR_CODE: MY-010116
 SUBSYSTEM: Server
      DATA: mysqld (mysqld 8.0.23) starting as process 96344
*************************** 2. row ***************************
```

```
    LOGGED: 2020-08-06 09:25:00.363521
 THREAD_ID: 1
      PRIO: System
ERROR_CODE: MY-013576
 SUBSYSTEM: InnoDB
      DATA: InnoDB initialization has started.
...
*************************** 65. row ***************************
    LOGGED: 2020-08-06 09:25:02.936146
 THREAD_ID: 0
      PRIO: Warning
ERROR_CODE: MY-010068
 SUBSYSTEM: Server
      DATA: CA certificate /var/mysql/sslinfo/cacert.pem is self signed.
...
*************************** 89. row ***************************
    LOGGED: 2020-08-06 09:25:03.112801
 THREAD_ID: 0
      PRIO: System
ERROR_CODE: MY-013292
 SUBSYSTEM: Server
      DATA: Admin interface ready for connections, address: '127.0.0.1' port: 33062
```

The `error_log` table has the following columns. As indicated in the descriptions, all but the `DATA` column correspond to fields of the underlying error event structure, which is described in Error Event Fields.

- `LOGGED`

  The event timestamp, with microsecond precision. `LOGGED` corresponds to the `time` field of error events, although with certain potential differences:

  - `time` values in the error log are displayed according to the `log_timestamps` system variable setting; see Early-Startup Logging Output Format.
  - The `LOGGED` column stores values using the `TIMESTAMP` data type, for which values are stored in UTC but displayed when retrieved in the current session time zone; see The DATE, DATETIME, and TIMESTAMP Types.

  To display `LOGGED` values in the same time zone as displayed in the error log file, first set the session time zone as follows:

  ```
  SET @@session.time_zone = @@global.log_timestamps;
  ```

  If the `log_timestamps` value is `UTC` and your system does not have named time zone support installed (see MySQL Server Time Zone Support), set the time zone like this:

  ```
  SET @@session.time_zone = '+00:00';
  ```

- `THREAD_ID`

  The MySQL thread ID. `THREAD_ID` corresponds to the `thread` field of error events.

  Within the Performance Schema, the `THREAD_ID` column in the `error_log` table is most similar to the `PROCESSLIST_ID` column of the `threads` table:

  - For foreground threads, `THREAD_ID` and `PROCESSLIST_ID` represent a connection identifier. This is the same value displayed in the `ID` column of the `INFORMATION_SCHEMA PROCESSLIST` table, displayed in the `Id` column of `SHOW PROCESSLIST` output, and returned by the `CONNECTION_ID()` function within the thread.
  - For background threads, `THREAD_ID` is 0 and `PROCESSLIST_ID` is `NULL`.

Many Performance Schema tables other than `error_log` has a column named `THREAD_ID`, but in those tables, the `THREAD_ID` column is a value assigned internally by the Performance Schema.

- `PRIO`

  The event priority. Permitted values are `System`, `Error`, `Warning`, `Note`. The `PRIO` column is based on the `label` field of error events, which itself is based on the underlying numeric `prio` field value.

- `ERROR_CODE`

  The numeric event error code. `ERROR_CODE` corresponds to the `error_code` field of error events.

- `SUBSYSTEM`

  The subsystem in which the event occurred. `SUBSYSTEM` corresponds to the `subsystem` field of error events.

- `DATA`

  The text representation of the error event. The format of this value depends on the format produced by the log sink component that generates the `error_log` row. For example, if the log sink is `log_sink_internal` or `log_sink_json`, `DATA` values represent error events in traditional or JSON format, respectively. (See Error Log Output Format.)

  Because the error log can be reconfigured to change the log sink component that supplies rows to the `error_log` table, and because different sinks produce different output formats, it is possible for rows written to the `error_log` table at different times to have different `DATA` formats.

The `error_log` table has these indexes:

- Primary key on (`LOGGED`)
- Index on (`THREAD_ID`)
- Index on (`PRIO`)
- Index on (`ERROR_CODE`)
- Index on (`SUBSYSTEM`)

`TRUNCATE TABLE` is not permitted for the `error_log` table.

Implementation and Configuration of the error_log Table

The Performance Schema `error_log` table is populated by error log sink components that write to the table in addition to writing formatted error events to the error log. Performance Schema support by log sinks has two parts:

- A log sink can write new error events to the `error_log` table as they occur.
- A log sink can provide a parser for extraction of previously written error messages. This enables a server instance to read messages written to an error log file by the previous instance and store them in the `error_log` table. Messages written during shutdown by the previous instance may be useful for diagnosing why shutdown occurred.

Currently, the traditional-format `log_sink_internal` and JSON-format `log_sink_json` sinks support writing new events to the `error_log` table and provide a parser for reading previously written error log files.

The log_error_services system variable controls which log components to enable for error logging. Its value is a pipeline of log filter and log sink components to be executed in left-to-right order when error events occur. The log_error_services value pertains to populating the error_log table as follows:

- At startup, the server examines the log_error_services value and chooses from it the leftmost log sink that satisfies these conditions:
  - A sink that supports the error_log table and provides a parser.
  - If none, a sink that supports the error_log table but provides no parser.

  If no log sink satisfies those conditions, the error_log table remains empty. Otherwise, if the sink provides a parser and log configuration enables a previously written error log file to be found, the server uses the sink parser to read the last part of the file and writes the old events it contains to the table. The sink then writes new error events to the table as they occur.
- At runtime, if the value of log_error_services changes, the server again examines it, this time looking for the leftmost enabled log sink that supports the error_log table, regardless of whether it provides a parser.

  If no such log sink exists, no additional error events are written to the error_log table. Otherwise, the newly configured sink writes new error events to the table as they occur.

Any configuration that affects output written to the error log affects error_log table contents. This includes settings such as those for verbosity, message suppression, and message filtering. It also applies to information read at startup from a previous log file. For example, messages not written during a previous server instance configured with low verbosity do not become available if the file is read by a current instance configured with higher verbosity.

The error_log table is a view on a fixed-size, in-memory ring buffer, with old events automatically discarded as necessary to make room for new ones. As shown in the following table, several status variables provide information about ongoing error_log operation.

Status Variable	Meaning
Error_log_buffered_bytes	Bytes used in table
Error_log_buffered_events	Events present in table
Error_log_expired_events	Events discarded from table
Error_log_latest_write	Time of last write to table

10.21.3 The host_cache Table

The MySQL server maintains an in-memory host cache that contains client host name and IP address information and is used to avoid Domain Name System (DNS) lookups. The host_cache table exposes the contents of this cache. The host_cache_size system variable controls the size of the host cache, as well as the size of the host_cache table. For operational and configuration information about the host cache, see DNS Lookups and the Host Cache.

Because the host_cache table exposes the contents of the host cache, it can be examined using SELECT statements. This may help you diagnose the causes of connection problems.

The host_cache table has these columns:

- IP

  The IP address of the client that connected to the server, expressed as a string.

- `HOST`

  The resolved DNS host name for that client IP, or `NULL` if the name is unknown.

- `HOST_VALIDATED`

  Whether the IP-to-host name-to-IP DNS resolution was performed successfully for the client IP. If `HOST_VALIDATED` is `YES`, the `HOST` column is used as the host name corresponding to the IP so that additional calls to DNS can be avoided. While `HOST_VALIDATED` is `NO`, DNS resolution is attempted for each connection attempt, until it eventually completes with either a valid result or a permanent error. This information enables the server to avoid caching bad or missing host names during temporary DNS failures, which would negatively affect clients forever.

- `SUM_CONNECT_ERRORS`

  The number of connection errors that are deemed “blocking” (assessed against the `max_connect_errors` system variable). Only protocol handshake errors are counted, and only for hosts that passed validation (`HOST_VALIDATED = YES`).

  Once `SUM_CONNECT_ERRORS` for a given host reaches the value of `max_connect_errors`, new connections from that host are blocked. The `SUM_CONNECT_ERRORS` value can exceed the `max_connect_errors` value because multiple connection attempts from a host can occur simultaneously while the host is not blocked. Any or all of them can fail, independently incrementing `SUM_CONNECT_ERRORS`, possibly beyond the value of `max_connect_errors`.

  Suppose that `max_connect_errors` is 200 and `SUM_CONNECT_ERRORS` for a given host is 199. If 10 clients attempt to connect from that host simultaneously, none of them are blocked because `SUM_CONNECT_ERRORS` has not reached 200. If blocking errors occur for five of the clients, `SUM_CONNECT_ERRORS` is increased by one for each client, for a resulting `SUM_CONNECT_ERRORS` value of 204. The other five clients succeed and are not blocked because the value of `SUM_CONNECT_ERRORS` when their connection attempts began had not reached 200. New connections from the host that begin after `SUM_CONNECT_ERRORS` reaches 200 are blocked.

- `COUNT_HOST_BLOCKED_ERRORS`

  The number of connections that were blocked because `SUM_CONNECT_ERRORS` exceeded the value of the `max_connect_errors` system variable.

- `COUNT_NAMEINFO_TRANSIENT_ERRORS`

  The number of transient errors during IP-to-host name DNS resolution.

- `COUNT_NAMEINFO_PERMANENT_ERRORS`

  The number of permanent errors during IP-to-host name DNS resolution.

- `COUNT_FORMAT_ERRORS`

  The number of host name format errors. MySQL does not perform matching of `Host` column values in the `mysql.user` system table against host names for which one or more of the initial components of the name are entirely numeric, such as `1.2.example.com`. The client IP address is used instead. For the rationale why this type of matching does not occur, see Specifying Account Names.

- `COUNT_ADDRINFO_TRANSIENT_ERRORS`

  The number of transient errors during host name-to-IP reverse DNS resolution.

- `COUNT_ADDRINFO_PERMANENT_ERRORS`

The number of permanent errors during host name-to-IP reverse DNS resolution.

- `COUNT_FCRDNS_ERRORS`

  The number of forward-confirmed reverse DNS errors. These errors occur when IP-to-host name-to-IP DNS resolution produces an IP address that does not match the client originating IP address.

- `COUNT_HOST_ACL_ERRORS`

  The number of errors that occur because no users are permitted to connect from the client host. In such cases, the server returns `ER_HOST_NOT_PRIVILEGED` and does not even ask for a user name or password.

- `COUNT_NO_AUTH_PLUGIN_ERRORS`

  The number of errors due to requests for an unavailable authentication plugin. A plugin can be unavailable if, for example, it was never loaded or a load attempt failed.

- `COUNT_AUTH_PLUGIN_ERRORS`

  The number of errors reported by authentication plugins.

  An authentication plugin can report different error codes to indicate the root cause of a failure. Depending on the type of error, one of these columns is incremented: `COUNT_AUTHENTICATION_ERRORS`, `COUNT_AUTH_PLUGIN_ERRORS`, `COUNT_HANDSHAKE_ERRORS`. New return codes are an optional extension to the existing plugin API. Unknown or unexpected plugin errors are counted in the `COUNT_AUTH_PLUGIN_ERRORS` column.

- `COUNT_HANDSHAKE_ERRORS`

  The number of errors detected at the wire protocol level.

- `COUNT_PROXY_USER_ERRORS`

  The number of errors detected when proxy user A is proxied to another user B who does not exist.

- `COUNT_PROXY_USER_ACL_ERRORS`

  The number of errors detected when proxy user A is proxied to another user B who does exist but for whom A does not have the `PROXY` privilege.

- `COUNT_AUTHENTICATION_ERRORS`

  The number of errors caused by failed authentication.

- `COUNT_SSL_ERRORS`

  The number of errors due to SSL problems.

- `COUNT_MAX_USER_CONNECTIONS_ERRORS`

  The number of errors caused by exceeding per-user connection quotas. See Setting Account Resource Limits.

- `COUNT_MAX_USER_CONNECTIONS_PER_HOUR_ERRORS`

  The number of errors caused by exceeding per-user connections-per-hour quotas. See Setting Account Resource Limits.

- COUNT_DEFAULT_DATABASE_ERRORS

  The number of errors related to the default database. For example, the database does not exist or the user has no privileges to access it.

- COUNT_INIT_CONNECT_ERRORS

  The number of errors caused by execution failures of statements in the init_connect system variable value.

- COUNT_LOCAL_ERRORS

  The number of errors local to the server implementation and not related to the network, authentication, or authorization. For example, out-of-memory conditions fall into this category.

- COUNT_UNKNOWN_ERRORS

  The number of other, unknown errors not accounted for by other columns in this table. This column is reserved for future use, in case new error conditions must be reported, and if preserving the backward compatibility and structure of the host_cache table is required.

- FIRST_SEEN

  The timestamp of the first connection attempt seen from the client in the IP column.

- LAST_SEEN

  The timestamp of the most recent connection attempt seen from the client in the IP column.

- FIRST_ERROR_SEEN

  The timestamp of the first error seen from the client in the IP column.

- LAST_ERROR_SEEN

  The timestamp of the most recent error seen from the client in the IP column.

The host_cache table has these indexes:

- Primary key on (IP)
- Index on (HOST)

TRUNCATE TABLE is permitted for the host_cache table. It requires the DROP privilege for the table. Truncating the table flushes the host cache, which has the effects described in Flushing the Host Cache.

10.21.4 The innodb_redo_log_files Table

The innodb_redo_log_files table contains a row for each active InnoDB redo log file. This table was introduced in MySQL 8.0.30.

The innodb_redo_log_files table has the following columns:

- FILE_ID

  The ID of the redo log file. The value corresponds to the redo log file number.

- FILE_NAME

The path and file name of the redo log file.

- `START_LSN`

  The log sequence number of the first block in the redo log file.

- `END_LSN`

  The log sequence number after the last block in the redo log file.

- `SIZE_IN_BYTES`

  The size of the redo log data in the file, in bytes. Data size is measured from the `END_LSN` to the start `>START_LSN`. The redo log file size on disk is slightly larger due to the file header (2048 bytes), which is not included in the value reported by this column.

- `IS_FULL`

  Whether the redo log file is full. A value of 0 indicates that free space in the file. A value of 1 indicates that the file is full.

- `CONSUMER_LEVEL`

  Reserved for future use.

10.21.5 The log_status Table

The `log_status` table provides information that enables an online backup tool to copy the required log files without locking those resources for the duration of the copy process.

When the `log_status` table is queried, the server blocks logging and related administrative changes for just long enough to populate the table, then releases the resources. The `log_status` table informs the online backup which point it should copy up to in the source's binary log and `gtid_executed` record, and the relay log for each replication channel. It also provides relevant information for individual storage engines, such as the last log sequence number (LSN) and the LSN of the last checkpoint taken for the `InnoDB` storage engine.

The `log_status` table has these columns:

- `SERVER_UUID`

  The server UUID for this server instance. This is the generated unique value of the read-only system variable `server_uuid`.

- `LOCAL`

  The log position state information from the source, provided as a single JSON object with the following keys:

`binary_log_file`	The name of the current binary log file.
`binary_log_position`	The current binary log position at the time the `log_status` table was accessed.
`gtid_executed`	The current value of the global server variable `gtid_executed` at the time the `log_status` table was accessed. This

information is consistent with the `binary_log_file` and `binary_log_position` keys.

- `REPLICATION`

  A JSON array of channels, each with the following information:

`channel_name`	The name of the replication channel. The default replication channel's name is the empty string (“”).
`relay_log_file`	The name of the current relay log file for the replication channel.
`relay_log_pos`	The current relay log position at the time the `log_status` table was accessed.

- `STORAGE_ENGINES`

  Relevant information from individual storage engines, provided as a JSON object with one key for each applicable storage engine.

The `log_status` table has no indexes.

The `BACKUP_ADMIN` privilege, as well as the `SELECT` privilege, is required for access to the `log_status` table.

`TRUNCATE TABLE` is not permitted for the `log_status` table.

10.21.6 The performance_timers Table

The `performance_timers` table shows which event timers are available:

```
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. For an explanation of how event timing occurs, see Section 5.1, “Performance Schema Event Timing”.

The `performance_timers` table has these columns:

- `TIMER_NAME`

  The timer name.

- `TIMER_FREQUENCY`

  The number of timer units per second. For a cycle timer, the frequency is generally related to the CPU speed. For example, on a system with a 2.4GHz processor, the `CYCLE` may be close to 2400000000.

- `TIMER_RESOLUTION`

  Indicates the number of timer units by which timer values increase. If a timer has a resolution of 10, its value increases by 10 each time.

- `TIMER_OVERHEAD`

  The minimal number of cycles of overhead to obtain one timing with the given timer. The Performance Schema determines this value by invoking the timer 20 times during initialization and picking the smallest value. The total overhead really is twice this amount because the instrumentation invokes the timer at the start and end of each event. The timer code is called only for timed events, so this overhead does not apply for nontimed events.

The `performance_timers` table has no indexes.

`TRUNCATE TABLE` is not permitted for the `performance_timers` table.

10.21.7 The processlist Table

The MySQL process list indicates the operations currently being performed by the set of threads executing within the server. The `processlist` table is one source of process information. For a comparison of this table with other sources, see Sources of Process Information.

The `processlist` table can be queried directly. If you have the `PROCESS` privilege, you can see all threads, even those belonging to other users. 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.

> **Note**
>
> If the `performance_schema_show_processlist` system variable is enabled, the `processlist` table also serves as the basis for an alternative implementation underlying the `SHOW PROCESSLIST` statement. For details, see later in this section.

The `processlist` table contains a row for each server process:

```
mysql> SELECT * FROM performance_schema.processlist\G
*************************** 1. row ***************************
     ID: 5
   USER: event_scheduler
   HOST: localhost
     DB: NULL
COMMAND: Daemon
   TIME: 137
  STATE: Waiting on empty queue
   INFO: NULL
*************************** 2. row ***************************
     ID: 9
   USER: me
   HOST: localhost:58812
     DB: NULL
COMMAND: Sleep
   TIME: 95
  STATE:
   INFO: NULL
*************************** 3. row ***************************
     ID: 10
   USER: me
   HOST: localhost:58834
     DB: test
COMMAND: Query
   TIME: 0
  STATE: executing
   INFO: SELECT * FROM performance_schema.processlist
...
```

The `processlist` 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 Using the Event Scheduler).

  > **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 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 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 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 Replication Threads.

- `STATE`

  An action, event, or state that indicates what the thread is doing. For descriptions of `STATE` values, see 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.

- `EXECUTION_ENGINE`

  The query execution engine. The value is either `PRIMARY` or `SECONDARY`. For use with HeatWave Service and HeatWave, where the `PRIMARY` engine is `InnoDB` and the `SECONDARY` engine is HeatWave (`RAPID`). For MySQL Community Edition Server, MySQL Enterprise Edition Server (on-premise), and HeatWave Service without HeatWave, the value is always `PRIMARY`. This column was added in MySQL 8.0.29.

The `processlist` table has these indexes:

- Primary key on (`ID`)

`TRUNCATE TABLE` is not permitted for the `processlist` table.

As mentioned previously, if the `performance_schema_show_processlist` system variable is enabled, the `processlist` table serves as the basis for an alternative implementation of other process information sources:

- The `SHOW PROCESSLIST` statement.
- The `mysqladmin processlist` command (which uses `SHOW PROCESSLIST` statement).

The default `SHOW PROCESSLIST` implementation iterates across active threads from within the thread manager while holding a global mutex. This has negative performance consequences, particularly on busy systems. The alternative `SHOW PROCESSLIST` implementation is based on the Performance Schema `processlist` table. This implementation queries active thread data from the Performance Schema rather than the thread manager and does not require a mutex.

MySQL configuration affects `processlist` table contents as follows:

- Minimum required configuration:
  - The MySQL server must be configured and built with thread instrumentation enabled. This is true by default; it is controlled using the `DISABLE_PSI_THREAD CMake` option.
  - The Performance Schema must be enabled at server startup. This is true by default; it is controlled using the `performance_schema` system variable.

  With that configuration satisfied, `performance_schema_show_processlist` enables or disables the alternative `SHOW PROCESSLIST` implementation. If the minimum configuration is not satisfied, the `processlist` table (and thus `SHOW PROCESSLIST`) may not return all data.
- Recommended configuration:
  - To avoid having some threads ignored:
    - Leave the `performance_schema_max_thread_instances` system variable set to its default or set it at least as great as the `max_connections` system variable.
    - Leave the `performance_schema_max_thread_classes` system variable set to its default.
  - To avoid having some `STATE` column values be empty, leave the `performance_schema_max_stage_classes` system variable set to its default.

  The default for those configuration parameters is `-1`, which causes the Performance Schema to autosize them at server startup. With the parameters set as indicated, the `processlist` table (and thus `SHOW PROCESSLIST`) produce complete process information.

The preceding configuration parameters affect the contents of the `processlist` table. For a given configuration, however, the `processlist` contents are unaffected by the `performance_schema_show_processlist` setting.

The alternative process list implementation does not apply to the `INFORMATION_SCHEMA PROCESSLIST` table or the `COM_PROCESS_INFO` command of the MySQL client/server protocol.

10.21.8 The threads Table

The `threads` table contains a row for each server thread. Each row contains information about a thread and indicates whether monitoring and historical event logging are enabled for it:

```
mysql> SELECT * FROM performance_schema.threads\G
*************************** 1. row ***************************
           THREAD_ID: 1
                NAME: thread/sql/main
                TYPE: BACKGROUND
      PROCESSLIST_ID: NULL
    PROCESSLIST_USER: NULL
    PROCESSLIST_HOST: NULL
      PROCESSLIST_DB: mysql
  PROCESSLIST_COMMAND: NULL
     PROCESSLIST_TIME: 418094
    PROCESSLIST_STATE: NULL
     PROCESSLIST_INFO: NULL
     PARENT_THREAD_ID: NULL
                ROLE: NULL
        INSTRUMENTED: YES
             HISTORY: YES
     CONNECTION_TYPE: NULL
        THREAD_OS_ID: 5856
      RESOURCE_GROUP: SYS_default
    EXECUTION_ENGINE: PRIMARY
   CONTROLLED_MEMORY: 1456
MAX_CONTROLLED_MEMORY: 67480
        TOTAL_MEMORY: 1270430
    MAX_TOTAL_MEMORY: 1307317
     TELEMETRY_ACTIVE: NO
...
```

When the Performance Schema initializes, it populates the `threads` table based on the threads in existence then. Thereafter, a new row is added each time the server creates a thread.

The `INSTRUMENTED` and `HISTORY` column values for new threads are determined by the contents of the `setup_actors` table. For information about how to use the `setup_actors` table to control these columns, see Section 5.6, “Pre-Filtering by Thread”.

Removal of rows from the `threads` table occurs when threads end. For a thread associated with a client session, removal occurs when the session ends. If a client has auto-reconnect enabled and the session reconnects after a disconnect, the session becomes associated with a new row in the `threads` table that has a different `PROCESSLIST_ID` value. The initial `INSTRUMENTED` and `HISTORY` values for the new thread may be different from those of the original thread: The `setup_actors` table may have changed in the meantime, and if the `INSTRUMENTED` or `HISTORY` value for the original thread was changed after the row was initialized, the change does not carry over to the new thread.

You can enable or disable thread monitoring (that is, whether events executed by the thread are instrumented) and historical event logging. To control the initial `INSTRUMENTED` and `HISTORY` values for new foreground threads, use the `setup_actors` table. To control these aspects of existing threads, set the `INSTRUMENTED` and `HISTORY` columns of `threads` table rows. (For more information about the conditions under which thread monitoring and historical event logging occur, see the descriptions of the `INSTRUMENTED` and `HISTORY` columns.)

For a comparison of the `threads` table columns with names having a prefix of `PROCESSLIST_` to other process information sources, see Sources of Process Information.

> **Important**
>
> For thread information sources other than the `threads` table, information about threads for other users is shown only if the current user has the `PROCESS` privilege. That is not true of the `threads` table; all rows are shown to any user who has the `SELECT` privilege for the table. Users who should not be able to see threads for other users by accessing the `threads` table should not be given the `SELECT` privilege for it.

The `threads` table has these columns:

- `THREAD_ID`

  A unique thread identifier.

- `NAME`

  The name associated with the thread instrumentation code in the server. For example, `thread/sql/one_connection` corresponds to the thread function in the code responsible for handling a user connection, and `thread/sql/main` stands for the `main()` function of the server.

- `TYPE`

  The thread type, either `FOREGROUND` or `BACKGROUND`. User connection threads are foreground threads. Threads associated with internal server activity are background threads. Examples are internal `InnoDB` threads, “binlog dump” threads sending information to replicas, and replication I/O and SQL threads.

- `PROCESSLIST_ID`

  For a foreground thread (associated with a user connection), this is the connection identifier. This is the same value displayed in the `ID` column of the `INFORMATION_SCHEMA PROCESSLIST` table, displayed in the `Id` column of `SHOW PROCESSLIST` output, and returned by the `CONNECTION_ID()` function within the thread.

  For a background thread (not associated with a user connection), `PROCESSLIST_ID` is `NULL`, so the values are not unique.

- `PROCESSLIST_USER`

  The user associated with a foreground thread, `NULL` for a background thread.

- `PROCESSLIST_HOST`

  The host name of the client associated with a foreground thread, `NULL` for a background thread.

  Unlike the `HOST` column of the `INFORMATION_SCHEMA PROCESSLIST` table or the `Host` column of `SHOW PROCESSLIST` output, the `PROCESSLIST_HOST` column does not include the port number for TCP/IP connections. To obtain this information from the Performance Schema, enable the socket instrumentation (which is not enabled by default) and examine the `socket_instances` table:

```
mysql> SELECT NAME, ENABLED, TIMED
       FROM performance_schema.setup_instruments
       WHERE NAME LIKE 'wait/io/socket%';
+----------------------------------------+---------+-------+
| NAME                                   | ENABLED | TIMED |
+----------------------------------------+---------+-------+
```

```
| wait/io/socket/sql/server_tcpip_socket | NO      | NO    |
| wait/io/socket/sql/server_unix_socket  | NO      | NO    |
| wait/io/socket/sql/client_connection   | NO      | NO    |
+----------------------------------------+---------+-------+
3 rows in set (0.01 sec)
mysql> UPDATE performance_schema.setup_instruments
       SET ENABLED='YES'
       WHERE NAME LIKE 'wait/io/socket%';
Query OK, 3 rows affected (0.00 sec)
Rows matched: 3  Changed: 3  Warnings: 0
mysql> SELECT * FROM performance_schema.socket_instances\G
*************************** 1. row ***************************
          EVENT_NAME: wait/io/socket/sql/client_connection
OBJECT_INSTANCE_BEGIN: 140612577298432
            THREAD_ID: 31
            SOCKET_ID: 53
                   IP: ::ffff:127.0.0.1
                 PORT: 55642
                STATE: ACTIVE
...
```

- `PROCESSLIST_DB`

  The default database for the thread, or `NULL` if none has been selected.

- `PROCESSLIST_COMMAND`

  For foreground threads, 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 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 Server Status Variables

  Background threads do not execute commands on behalf of clients, so this column may be `NULL`.

- `PROCESSLIST_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 Replication Threads.

- `PROCESSLIST_STATE`

  An action, event, or state that indicates what the thread is doing. For descriptions of `PROCESSLIST_STATE` values, see Examining Server Thread (Process) Information. If the value if `NULL`, the thread may correspond to an idle client session or the work it is doing is not instrumented with stages.

  Most states correspond to very quick operations. If a thread stays in a given state for many seconds, there might be a problem that bears investigation.

- `PROCESSLIST_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 `PROCESSLIST_INFO` value shows the `SELECT` statement.

- `PARENT_THREAD_ID`

  If this thread is a subthread (spawned by another thread), this is the `THREAD_ID` value of the spawning thread.

- `ROLE`

  Unused.

- `INSTRUMENTED`

  Whether events executed by the thread are instrumented. The value is `YES` or `NO`.

  - For foreground threads, the initial `INSTRUMENTED` value is determined by whether the user account associated with the thread matches any row in the `setup_actors` table. Matching is based on the values of the `PROCESSLIST_USER` and `PROCESSLIST_HOST` columns.

    If the thread spawns a subthread, matching occurs again for the `threads` table row created for the subthread.

  - For background threads, `INSTRUMENTED` is `YES` by default. `setup_actors` is not consulted because there is no associated user for background threads.

  - For any thread, its `INSTRUMENTED` value can be changed during the lifetime of the thread.

  For monitoring of events executed by the thread to occur, 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 have the `ENABLED` column set to `YES` in the `setup_instruments` table.

- `HISTORY`

  Whether to log historical events for the thread. The value is `YES` or `NO`.

  - For foreground threads, the initial `HISTORY` value is determined by whether the user account associated with the thread matches any row in the `setup_actors` table. Matching is based on the values of the `PROCESSLIST_USER` and `PROCESSLIST_HOST` columns.

    If the thread spawns a subthread, matching occurs again for the `threads` table row created for the subthread.

  - For background threads, `HISTORY` is `YES` by default. `setup_actors` is not consulted because there is no associated user for background threads.

  - For any thread, its `HISTORY` value can be changed during the lifetime of the thread.

  For historical event logging for the thread 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 have the `ENABLED` column set to `YES` in the `setup_instruments` table.

- `CONNECTION_TYPE`

The protocol used to establish the connection, or `NULL` for background threads. Permitted values are `TCP/IP` (TCP/IP connection established without encryption), `SSL/TLS` (TCP/IP connection established with encryption), `Socket` (Unix socket file connection), `Named Pipe` (Windows named pipe connection), and `Shared Memory` (Windows shared memory connection).

- `THREAD_OS_ID`

  The thread or task identifier as defined by the underlying operating system, if there is one:

  - When a MySQL thread is associated with the same operating system thread for its lifetime, `THREAD_OS_ID` contains the operating system thread ID.
  - When a MySQL thread is not associated with the same operating system thread for its lifetime, `THREAD_OS_ID` contains `NULL`. This is typical for user sessions when the thread pool plugin is used (see MySQL Enterprise Thread Pool).

  For Windows, `THREAD_OS_ID` corresponds to the thread ID visible in Process Explorer (https://technet.microsoft.com/en-us/sysinternals/bb896653.aspx).

  For Linux, `THREAD_OS_ID` corresponds to the value of the `gettid()` function. This value is exposed, for example, using the `perf` or `ps -L` commands, or in the `proc` file system (`/proc/[pid]/task/[tid]`). For more information, see the `perf-stat(1)`, `ps(1)`, and `proc(5)` man pages.

- `RESOURCE_GROUP`

  The resource group label. This value is `NULL` if resource groups are not supported on the current platform or server configuration (see Resource Group Restrictions).

- `EXECUTION_ENGINE`

  The query execution engine. The value is either `PRIMARY` or `SECONDARY`. For use with HeatWave Service and HeatWave, where the `PRIMARY` engine is `InnoDB` and the `SECONDARY` engine is HeatWave (`RAPID`). For MySQL Community Edition Server, MySQL Enterprise Edition Server (on-premise), and HeatWave Service without HeatWave, the value is always `PRIMARY`. This column was added in MySQL 8.0.29.

- `CONTROLLED_MEMORY`

  Amount of controlled memory used by the thread.

  This column was added in MySQL 8.0.31.

- `MAX_CONTROLLED_MEMORY`

  Maximum value of `CONTROLLED_MEMORY` seen during the thread execution.

  This column was added in MySQL 8.0.31.

- `TOTAL_MEMORY`

  The current amount of memory, controlled or not, used by the thread.

  This column was added in MySQL 8.0.31.

- `MAX_TOTAL_MEMORY`

  The maximum value of `TOTAL_MEMORY` seen during the thread execution.

This column was added in MySQL 8.0.31.

- `TELEMETRY_ACTIVE`

  Whether the thread has an active telemetry seesion attached. The value is `YES` or `NO`.

  This column was added in MySQL 8.0.33.

The `threads` table has these indexes:

- Primary key on (`THREAD_ID`)
- Index on (`NAME`)
- Index on (`PROCESSLIST_ID`)
- Index on (`PROCESSLIST_USER`, `PROCESSLIST_HOST`)
- Index on (`PROCESSLIST_HOST`)
- Index on (`THREAD_OS_ID`)
- Index on (`RESOURCE_GROUP`)

`TRUNCATE TABLE` is not permitted for the `threads` table.

10.21.9 The tls_channel_status Table

Connection interface TLS properties are set at server startup, and can be updated at runtime using the `ALTER INSTANCE RELOAD TLS` statement. See Server-Side Runtime Configuration and Monitoring for Encrypted Connections.

The `tls_channel_status` table (available as of MySQL 8.0.21) provides information about connection interface TLS properties:

```
mysql> SELECT * FROM performance_schema.tls_channel_status\G
*************************** 1. row ***************************
 CHANNEL: mysql_main
PROPERTY: Enabled
   VALUE: Yes
*************************** 2. row ***************************
 CHANNEL: mysql_main
PROPERTY: ssl_accept_renegotiates
   VALUE: 0
*************************** 3. row ***************************
 CHANNEL: mysql_main
PROPERTY: Ssl_accepts
   VALUE: 2
...
*************************** 29. row ***************************
 CHANNEL: mysql_admin
PROPERTY: Enabled
   VALUE: No
*************************** 30. row ***************************
 CHANNEL: mysql_admin
PROPERTY: ssl_accept_renegotiates
   VALUE: 0
*************************** 31. row ***************************
 CHANNEL: mysql_admin
PROPERTY: Ssl_accepts
   VALUE: 0
...
```

The `tls_channel_status` table has these columns:

- `CHANNEL`

  The name of the connection interface to which the TLS property row applies. `mysql_main` and `mysql_admin` are the channel names for the main and administrative connection interfaces, respectively. For information about the different interfaces, see Connection Interfaces.

- `PROPERTY`

  The TLS property name. The row for the `Enabled` property indicates overall interface status, where the interface and its status are named in the `CHANNEL` and `VALUE` columns, respectively. Other property names indicate particular TLS properties. These often correspond to the names of TLS-related status variables.

- `VALUE`

  The TLS property value.

The properties exposed by this table are not fixed and depend on the instrumentation implemented by each channel.

For each channel, the row with a `PROPERTY` value of `Enabled` indicates whether the channel supports encrypted connections, and other channel rows indicate TLS context properties:

- For `mysql_main`, the `Enabled` property is `yes` or `no` to indicate whether the main interface supports encrypted connections. Other channel rows display TLS context properties for the main interface.

  For the main interface, similar status information can be obtained using these statements:

  ```
  SHOW GLOBAL STATUS LIKE 'current_tls%';
  SHOW GLOBAL STATUS LIKE 'ssl%';
  ```

- For `mysql_admin`, the `Enabled` property is `no` if the administrative interface is not enabled or it is enabled but does not support encrypted connections. `Enabled` is `yes` if the interface is enabled and supports encrypted connections.

  When `Enabled` is `yes`, the other `mysql_admin` rows indicate channel properties for the administrative interface TLS context only if some nondefault TLS parameter value is configured for that interface. (This is the case if any `admin_tls_`*`xxx`* or `admin_ssl_`*`xxx`* system variable is set to a value different from its default.) Otherwise, the administrative interface uses the same TLS context as the main interface.

The `tls_channel_status` table has no indexes.

`TRUNCATE TABLE` is not permitted for the `tls_channel_status` table.

10.21.10 The user_defined_functions Table

The `user_defined_functions` table contains a row for each loadable function registered automatically by a component or plugin, or manually by a `CREATE FUNCTION` statement. For information about operations that add or remove table rows, see Installing and Uninstalling Loadable Functions.

> **Note**
>
> The name of the `user_defined_functions` table stems from the terminology used at its inception for the type of function now known as a loadable function (that is, user-defined function, or UDF).

The `user_defined_functions` table has these columns:

- `UDF_NAME`

  The function name as referred to in SQL statements. The value is `NULL` if the function was registered by a `CREATE FUNCTION` statement and is in the process of unloading.

- `UDF_RETURN_TYPE`

  The function return value type. The value is one of `int`, `decimal`, `real`, `char`, or `row`.

- `UDF_TYPE`

  The function type. The value is one of `function` (scalar) or `aggregate`.

- `UDF_LIBRARY`

  The name of the library file containing the executable function code. The file is located in the directory named by the `plugin_dir` system variable. The value is `NULL` if the function was registered by a component or plugin rather than by a `CREATE FUNCTION` statement.

- `UDF_USAGE_COUNT`

  The current function usage count. This is used to tell whether statements currently are accessing the function.

The `user_defined_functions` table has these indexes:

- Primary key on (`UDF_NAME`)

`TRUNCATE TABLE` is not permitted for the `user_defined_functions` table.

The `mysql.func` system table also lists installed loadable functions, but only those installed using `CREATE FUNCTION`. The `user_defined_functions` table lists loadable functions installed using `CREATE FUNCTION` as well as loadable functions installed automatically by components or plugins. This difference makes `user_defined_functions` preferable to `mysql.func` for checking which loadable functions are installed.

Removing a plugin with `UNINSTALL PLUGIN` does not affect information already collected for code in that plugin. Time spent executing the code while the plugin was loaded was still spent even if the plugin is unloaded later. The associated event information, including aggregate information, remains readable in `performance_schema` database tables. For additional information about the effect of plugin installation and removal, see Chapter 8, *Performance Schema Status Monitoring*.

A plugin implementor who instruments plugin code should document its instrumentation characteristics to enable those who load the plugin to account for its requirements. For example, a third-party storage engine should include in its documentation how much memory the engine needs for mutex and other instruments.

Chapter 12 Performance Schema System Variables

The Performance Schema implements several system variables that provide configuration information:

```
mysql> SHOW VARIABLES LIKE 'perf%';
+----------------------------------------------------+-------+
| Variable_name                                      | Value |
+----------------------------------------------------+-------+
| performance_schema                                 | ON    |
| performance_schema_accounts_size                   | -1    |
| performance_schema_digests_size                    | 10000 |
| 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_transactions_history_long_size | 10000 |
| performance_schema_events_transactions_history_size | 10    |
| performance_schema_events_waits_history_long_size  | 10000 |
| performance_schema_events_waits_history_size       | 10    |
| performance_schema_hosts_size                      | -1    |
| performance_schema_max_cond_classes                | 80    |
| performance_schema_max_cond_instances              | -1    |
| performance_schema_max_digest_length               | 1024  |
| performance_schema_max_file_classes                | 50    |
| performance_schema_max_file_handles                | 32768 |
| performance_schema_max_file_instances              | -1    |
| performance_schema_max_index_stat                  | -1    |
| performance_schema_max_memory_classes              | 320   |
| performance_schema_max_metadata_locks              | -1    |
| performance_schema_max_mutex_classes               | 350   |
| performance_schema_max_mutex_instances             | -1    |
| performance_schema_max_prepared_statements_instances | -1    |
| performance_schema_max_program_instances           | -1    |
| performance_schema_max_rwlock_classes              | 40    |
| performance_schema_max_rwlock_instances            | -1    |
| performance_schema_max_socket_classes              | 10    |
| performance_schema_max_socket_instances            | -1    |
| performance_schema_max_sql_text_length             | 1024  |
| performance_schema_max_stage_classes               | 150   |
| performance_schema_max_statement_classes           | 192   |
| performance_schema_max_statement_stack             | 10    |
| performance_schema_max_table_handles               | -1    |
| performance_schema_max_table_instances             | -1    |
| performance_schema_max_table_lock_stat             | -1    |
| performance_schema_max_thread_classes              | 50    |
| performance_schema_max_thread_instances            | -1    |
| performance_schema_session_connect_attrs_size      | 512   |
| performance_schema_setup_actors_size               | -1    |
| performance_schema_setup_objects_size              | -1    |
| performance_schema_users_size                      | -1    |
+----------------------------------------------------+-------+
```

Performance Schema system variables can be set at server startup on the command line or in option files, and many can be set at runtime. See Performance Schema Option and Variable Reference.

The Performance Schema automatically sizes the values of several of its parameters at server startup if they are not set explicitly. For more information, see Chapter 4, *Performance Schema Startup Configuration*.

Performance Schema system variables have the following meanings:

- `performance_schema`

Command-Line Format	`--performance-schema[={OFF|ON}]`

System Variable	performance_schema
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Boolean
Default Value	ON

The value of this variable is ON or OFF to indicate whether the Performance Schema is enabled. By default, the value is ON. At server startup, you can specify this variable with no value or a value of ON or 1 to enable it, or with a value of OFF or 0 to disable it.

Even when the Performance Schema is disabled, it continues to populate the global_variables, session_variables, global_status, and session_status tables. This occurs as necessary to permit the results for the SHOW VARIABLES and SHOW STATUS statements to be drawn from those tables. The Performance Schema also populates some of the replication tables when disabled.

- performance_schema_accounts_size

Command-Line Format	--performance-schema-accounts-size=#
System Variable	performance_schema_accounts_size
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autoscaling; do not assign this literal value)
Minimum Value	-1 (signifies autoscaling; do not assign this literal value)
Maximum Value	1048576

The number of rows in the accounts table. If this variable is 0, the Performance Schema does not maintain connection statistics in the accounts table or status variable information in the status_by_account table.

- performance_schema_digests_size

Command-Line Format	--performance-schema-digests-size=#
System Variable	performance_schema_digests_size
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autosizing; do not assign this literal value)
Minimum Value	-1 (signifies autoscaling; do not assign this literal value)

Maximum Value	1048576

The maximum number of rows in the events_statements_summary_by_digest table. If this maximum is exceeded such that a digest cannot be instrumented, the Performance Schema increments the Performance_schema_digest_lost status variable.

For more information about statement digesting, see Performance Schema Statement Digests and Sampling.

- performance_schema_error_size

Command-Line Format	--performance-schema-error-size=#
System Variable	performance_schema_error_size
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	number of server error codes
Minimum Value	0
Maximum Value	1048576

The number of instrumented server error codes. The default value is the actual number of server error codes. Although the value can be set anywhere from 0 to its maximum, the intended use is to set it to either its default (to instrument all errors) or 0 (to instrument no errors).

Error information is aggregated in summary tables; see Section 10.20.11, “Error Summary Tables”. If an error occurs that is not instrumented, information for the occurrence is aggregated to the NULL row in each summary table; that is, to the row with ERROR_NUMBER=0, ERROR_NAME=NULL, and SQLSTATE=NULL.

- performance_schema_events_stages_history_long_size

Command-Line Format	--performance-schema-events-stages-history-long-size=#
System Variable	performance_schema_events_stages_history_long
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autosizing; do not assign this literal value)
Minimum Value	-1 (signifies autoscaling; do not assign this literal value)
Maximum Value	1048576

The number of rows in the events_stages_history_long table.

- `performance_schema_events_stages_history_size`

Command-Line Format	`--performance-schema-events-stages-history-size=#`
System Variable	`performance_schema_events_stages_history_size`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autosizing; do not assign this literal value)
Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`1024`

  The number of rows per thread in the `events_stages_history` table.

- `performance_schema_events_statements_history_long_size`

Command-Line Format	`--performance-schema-events-statements-history-long-size=#`
System Variable	`performance_schema_events_statements_history_lo`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autosizing; do not assign this literal value)
Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`1048576`

  The number of rows in the `events_statements_history_long` table.

- `performance_schema_events_statements_history_size`

Command-Line Format	`--performance-schema-events-statements-history-size=#`
System Variable	`performance_schema_events_statements_history_si`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autosizing; do not assign this literal value)

Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`1024`

The number of rows per thread in the `events_statements_history` table.

- `performance_schema_events_transactions_history_long_size`

Command-Line Format	`--performance-schema-events-transactions-history-long-size=#`
System Variable	`performance_schema_events_transactions_histo`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autosizing; do not assign this literal value)
Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`1048576`

The number of rows in the `events_transactions_history_long` table.

- `performance_schema_events_transactions_history_size`

Command-Line Format	`--performance-schema-events-transactions-history-size=#`
System Variable	`performance_schema_events_transactions_histo`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autosizing; do not assign this literal value)
Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`1024`

The number of rows per thread in the `events_transactions_history` table.

- `performance_schema_events_waits_history_long_size`

Command-Line Format	`--performance-schema-events-waits-history-long-size=#`
System Variable	`performance_schema_events_waits_history_long`
Scope	Global
Dynamic	No

SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autosizing; do not assign this literal value)
Minimum Value	-1 (signifies autoscaling; do not assign this literal value)
Maximum Value	1048576

The number of rows in the events_waits_history_long table.

- performance_schema_events_waits_history_size

Command-Line Format	--performance-schema-events-waits-history-size=#
System Variable	performance_schema_events_waits_history_size
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autosizing; do not assign this literal value)
Minimum Value	-1 (signifies autoscaling; do not assign this literal value)
Maximum Value	1024

The number of rows per thread in the events_waits_history table.

- performance_schema_hosts_size

Command-Line Format	--performance-schema-hosts-size=#
System Variable	performance_schema_hosts_size
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autoscaling; do not assign this literal value)
Minimum Value	-1 (signifies autoscaling; do not assign this literal value)
Maximum Value	1048576

The number of rows in the hosts table. If this variable is 0, the Performance Schema does not maintain connection statistics in the hosts table or status variable information in the status_by_host table.

- `performance_schema_max_cond_classes`

Command-Line Format	`--performance-schema-max-cond-classes=#`
System Variable	`performance_schema_max_cond_classes`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value (≥ 8.0.27)	`150`
Default Value (≥ 8.0.13, ≤ 8.0.26)	`100`
Default Value (≤ 8.0.12)	`80`
Minimum Value	`0`
Maximum Value (≥ 8.0.12)	`1024`
Maximum Value (8.0.11)	`256`

The maximum number of condition instruments. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- `performance_schema_max_cond_instances`

Command-Line Format	`--performance-schema-max-cond-instances=#`
System Variable	`performance_schema_max_cond_instances`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autoscaling; do not assign this literal value)
Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`1048576`

The maximum number of instrumented condition objects. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- `performance_schema_max_digest_length`

Command-Line Format	`--performance-schema-max-digest-length=#`
System Variable	`performance_schema_max_digest_length`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No

Type	Integer
Default Value	1024
Minimum Value	0
Maximum Value	1048576
Unit	bytes

The maximum number of bytes of memory reserved per statement for computation of normalized statement digest values in the Performance Schema. This variable is related to max_digest_length; see the description of that variable in Server System Variables.

For more information about statement digesting, including considerations regarding memory use, see Performance Schema Statement Digests and Sampling.

- performance_schema_max_digest_sample_age

Command-Line Format	--performance-schema-max-digest-sample-age=#
System Variable	performance_schema_max_digest_sample_age
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	60
Minimum Value	0
Maximum Value	1048576
Unit	seconds

This variable affects statement sampling for the events_statements_summary_by_digest table. When a new table row is inserted, the statement that produced the row digest value is stored as the current sample statement associated with the digest. Thereafter, when the server sees other statements with the same digest value, it determines whether to use the new statement to replace the current sample statement (that is, whether to resample). Resampling policy is based on the comparative wait times of the current sample statement and new statement and, optionally, the age of the current sample statement:

- Resampling based on wait times: If the new statement wait time has a wait time greater than that of the current sample statement, it becomes the current sample statement.
- Resampling based on age: If the performance_schema_max_digest_sample_age system variable has a value greater than zero and the current sample statement is more than that many seconds old, the current statement is considered “too old” and the new statement replaces it. This occurs even if the new statement wait time is less than that of the current sample statement.

For information about statement sampling, see Performance Schema Statement Digests and Sampling.

- performance_schema_max_file_classes

Command-Line Format	--performance-schema-max-file-classes=#

System Variable	performance_schema_max_file_classes
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	80
Minimum Value	0
Maximum Value (≥ 8.0.12)	1024
Maximum Value (8.0.11)	256

The maximum number of file instruments. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- performance_schema_max_file_handles

Command-Line Format	--performance-schema-max-file-handles=#
System Variable	performance_schema_max_file_handles
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	32768
Minimum Value	0
Maximum Value	1048576

The maximum number of opened file objects. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

The value of performance_schema_max_file_handles should be greater than the value of open_files_limit: open_files_limit affects the maximum number of open file handles the server can support and performance_schema_max_file_handles affects how many of these file handles can be instrumented.

- performance_schema_max_file_instances

Command-Line Format	--performance-schema-max-file-instances=#
System Variable	performance_schema_max_file_instances
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autoscaling; do not assign this literal value)

Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`1048576`

The maximum number of instrumented file objects. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- `performance_schema_max_index_stat`

Command-Line Format	`--performance-schema-max-index-stat=#`
System Variable	`performance_schema_max_index_stat`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autosizing; do not assign this literal value)
Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`1048576`

The maximum number of indexes for which the Performance Schema maintains statistics. If this maximum is exceeded such that index statistics are lost, the Performance Schema increments the `Performance_schema_index_stat_lost` status variable. The default value is autosized using the value of `performance_schema_max_table_instances`.

- `performance_schema_max_memory_classes`

Command-Line Format	`--performance-schema-max-memory-classes=#`
System Variable	`performance_schema_max_memory_classes`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`450`
Minimum Value	`0`
Maximum Value	`1024`

The maximum number of memory instruments. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- `performance_schema_max_metadata_locks`

Command-Line Format	`--performance-schema-max-metadata-locks=#`
System Variable	`performance_schema_max_metadata_locks`

Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autoscaling; do not assign this literal value)
Minimum Value	-1 (signifies autoscaling; do not assign this literal value)
Maximum Value	10485760

The maximum number of metadata lock instruments. This value controls the size of the metadata_locks table. If this maximum is exceeded such that a metadata lock cannot be instrumented, the Performance Schema increments the Performance_schema_metadata_lock_lost status variable.

- performance_schema_max_mutex_classes

Command-Line Format	--performance-schema-max-mutex-classes=#
System Variable	performance_schema_max_mutex_classes
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value (≥ 8.0.27)	350
Default Value (≥ 8.0.12, ≤ 8.0.26)	300
Default Value (8.0.11)	250
Minimum Value	0
Maximum Value (≥ 8.0.12)	1024
Maximum Value (8.0.11)	256

The maximum number of mutex instruments. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- performance_schema_max_mutex_instances

Command-Line Format	--performance-schema-max-mutex-instances=#
System Variable	performance_schema_max_mutex_instances
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autoscaling; do not assign this literal value)

Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`104857600`

The maximum number of instrumented mutex objects. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- `performance_schema_max_prepared_statements_instances`

Command-Line Format	`--performance-schema-max-prepared-statements-instances=#`
System Variable	`performance_schema_max_prepared_statements_inst`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autoscaling; do not assign this literal value)
Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`4194304`

The maximum number of rows in the `prepared_statements_instances` table. If this maximum is exceeded such that a prepared statement cannot be instrumented, the Performance Schema increments the `Performance_schema_prepared_statements_lost` status variable. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

The default value of this variable is autosized based on the value of the `max_prepared_stmt_count` system variable.

- `performance_schema_max_rwlock_classes`

Command-Line Format	`--performance-schema-max-rwlock-classes=#`
System Variable	`performance_schema_max_rwlock_classes`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value (≥ 8.0.12)	`100`
Default Value (8.0.11)	`60`
Minimum Value	`0`
Maximum Value (≥ 8.0.12)	`1024`
Maximum Value (8.0.11)	`256`

The maximum number of rwlock instruments. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- `performance_schema_max_program_instances`

Command-Line Format	`--performance-schema-max-program-instances=#`
System Variable	`performance_schema_max_program_instances`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autoscaling; do not assign this literal value)
Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`1048576`

The maximum number of stored programs for which the Performance Schema maintains statistics. If this maximum is exceeded, the Performance Schema increments the `Performance_schema_program_lost` status variable. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- `performance_schema_max_rwlock_instances`

Command-Line Format	`--performance-schema-max-rwlock-instances=#`
System Variable	`performance_schema_max_rwlock_instances`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autosizing; do not assign this literal value)
Minimum Value	`-1` (signifies autosizing; do not assign this literal value)
Maximum Value	`104857600`

The maximum number of instrumented rwlock objects. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- `performance_schema_max_socket_classes`

Command-Line Format	`--performance-schema-max-socket-classes=#`
System Variable	`performance_schema_max_socket_classes`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No

Type	Integer
Default Value	10
Minimum Value	0
Maximum Value (≥ 8.0.12)	1024
Maximum Value (8.0.11)	256

The maximum number of socket instruments. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- performance_schema_max_socket_instances

Command-Line Format	--performance-schema-max-socket-instances=#
System Variable	performance_schema_max_socket_instances
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autoscaling; do not assign this literal value)
Minimum Value	-1 (signifies autoscaling; do not assign this literal value)
Maximum Value	1048576

The maximum number of instrumented socket objects. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- performance_schema_max_sql_text_length

Command-Line Format	--performance-schema-max-sql-text-length=#
System Variable	performance_schema_max_sql_text_length
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	1024
Minimum Value	0
Maximum Value	1048576
Unit	bytes

The maximum number of bytes used to store SQL statements. The value applies to storage required for these columns:

- The SQL_TEXT column of the events_statements_current, events_statements_history, and events_statements_history_long statement event tables.

- The `QUERY_SAMPLE_TEXT` column of the `events_statements_summary_by_digest` summary table.

Any bytes in excess of `performance_schema_max_sql_text_length` are discarded and do not appear in the column. Statements differing only after that many initial bytes are indistinguishable in the column.

Decreasing the `performance_schema_max_sql_text_length` value reduces memory use but causes more statements to become indistinguishable if they differ only at the end. Increasing the value increases memory use but permits longer statements to be distinguished.

- `performance_schema_max_stage_classes`

Command-Line Format	`--performance-schema-max-stage-classes=#`
System Variable	`performance_schema_max_stage_classes`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value (≥ 8.0.13)	`175`
Default Value (≤ 8.0.12)	`150`
Minimum Value	`0`
Maximum Value (≥ 8.0.12)	`1024`
Maximum Value (8.0.11)	`256`

The maximum number of stage instruments. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- `performance_schema_max_statement_classes`

Command-Line Format	`--performance-schema-max-statement-classes=#`
System Variable	`performance_schema_max_statement_classes`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Minimum Value	`0`

Maximum Value	256

The maximum number of statement instruments. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

The default value is calculated at server build time based on the number of commands in the client/server protocol and the number of SQL statement types supported by the server.

This variable should not be changed, unless to set it to 0 to disable all statement instrumentation and save all memory associated with it. Setting the variable to nonzero values other than the default has no benefit; in particular, values larger than the default cause more memory to be allocated then is needed.

- `performance_schema_max_statement_stack`

Command-Line Format	`--performance-schema-max-statement-stack=#`
System Variable	`performance_schema_max_statement_stack`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	10
Minimum Value	1
Maximum Value	256

The maximum depth of nested stored program calls for which the Performance Schema maintains statistics. When this maximum is exceeded, the Performance Schema increments the `Performance_schema_nested_statement_lost` status variable for each stored program statement executed.

- `performance_schema_max_table_handles`

Command-Line Format	`--performance-schema-max-table-handles=#`
System Variable	`performance_schema_max_table_handles`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	-1 (signifies autoscaling; do not assign this literal value)
Minimum Value	-1 (signifies autoscaling; do not assign this literal value)
Maximum Value	1048576

The maximum number of opened table objects. This value controls the size of the `table_handles` table. If this maximum is exceeded such that a table handle cannot be instrumented, the Performance Schema increments the `Performance_schema_table_handles_lost` status variable. For

information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- `performance_schema_max_table_instances`

Command-Line Format	`--performance-schema-max-table-instances=#`
System Variable	`performance_schema_max_table_instances`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autoscaling; do not assign this literal value)
Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`1048576`

The maximum number of instrumented table objects. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- `performance_schema_max_table_lock_stat`

Command-Line Format	`--performance-schema-max-table-lock-stat=#`
System Variable	`performance_schema_max_table_lock_stat`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autosizing; do not assign this literal value)
Minimum Value	`-1` (signifies autoscaling; do not assign this literal value)
Maximum Value	`1048576`

The maximum number of tables for which the Performance Schema maintains lock statistics. If this maximum is exceeded such that table lock statistics are lost, the Performance Schema increments the `Performance_schema_table_lock_stat_lost` status variable.

- `performance_schema_max_thread_classes`

Command-Line Format	`--performance-schema-max-thread-classes=#`
System Variable	`performance_schema_max_thread_classes`
Scope	Global
Dynamic	No

SET_VAR Hint Applies	No
Type	Integer
Default Value	100
Minimum Value	0
Maximum Value (≥ 8.0.12)	1024
Maximum Value (8.0.11)	256

The maximum number of thread instruments. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

- performance_schema_max_thread_instances

Command-Line Format	--performance-schema-max-thread-instances=#
System Variable	performance_schema_max_thread_instances
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autosizing; do not assign this literal value)
Minimum Value	-1 (signifies autoscaling; do not assign this literal value)
Maximum Value	1048576

The maximum number of instrumented thread objects. The value controls the size of the threads table. If this maximum is exceeded such that a thread cannot be instrumented, the Performance Schema increments the Performance_schema_thread_instances_lost status variable. For information about how to set and use this variable, see Chapter 8, *Performance Schema Status Monitoring*.

The max_connections system variable affects how many threads can run in the server. performance_schema_max_thread_instances affects how many of these running threads can be instrumented.

The variables_by_thread and status_by_thread tables contain system and status variable information only about foreground threads. If not all threads are instrumented by the Performance Schema, this table misses some rows. In this case, the Performance_schema_thread_instances_lost status variable is greater than zero.

- performance_schema_session_connect_attrs_size

Command-Line Format	--performance-schema-session-connect-attrs-size=#
System Variable	performance_schema_session_connect_attrs_size
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No

Type	Integer
Default Value	`-1` (signifies autosizing; do not assign this literal value)
Minimum Value	`-1` (signifies autosizing; do not assign this literal value)
Maximum Value	`1048576`
Unit	bytes

The amount of preallocated memory per thread reserved to hold connection attribute key-value pairs. If the aggregate size of connection attribute data sent by a client is larger than this amount, the Performance Schema truncates the attribute data, increments the `Performance_schema_session_connect_attrs_lost` status variable, and writes a message to the error log indicating that truncation occurred if the `log_error_verbosity` system variable is greater than 1. A `_truncated` attribute is also added to the session attributes with a value indicating how many bytes were lost, if the attribute buffer has sufficient space. This enables the Performance Schema to expose per-connection truncation information in the connection attribute tables. This information can be examined without having to check the error log.

The default value of `performance_schema_session_connect_attrs_size` is autosized at server startup. This value may be small, so if truncation occurs (`Performance_schema_session_connect_attrs_lost` becomes nonzero), you may wish to set `performance_schema_session_connect_attrs_size` explicitly to a larger value.

Although the maximum permitted `performance_schema_session_connect_attrs_size` value is 1MB, the effective maximum is 64KB because the server imposes a limit of 64KB on the aggregate size of connection attribute data it accepts. If a client attempts to send more than 64KB of attribute data, the server rejects the connection. For more information, see Section 10.9, “Performance Schema Connection Attribute Tables”.

- `performance_schema_setup_actors_size`

Command-Line Format	`--performance-schema-setup-actors-size=#`
System Variable	`performance_schema_setup_actors_size`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`-1` (signifies autoscaling; do not assign this literal value)
Minimum Value	`-1` (signifies autosizing; do not assign this literal value)
Maximum Value	`1048576`

The number of rows in the `setup_actors` table.

- `performance_schema_setup_objects_size`

Command-Line Format	`--performance-schema-setup-objects-size=#`

System Variable	performance_schema_setup_objects_size
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autoscaling; do not assign this literal value)
Minimum Value	-1 (signifies autoscaling; do not assign this literal value)
Maximum Value	1048576

The number of rows in the setup_objects table.

- performance_schema_show_processlist

Command-Line Format	--performance-schema-show-processlist[={OFF|ON}]
Introduced	8.0.22
Deprecated	8.0.35
System Variable	performance_schema_show_processlist
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	OFF

The SHOW PROCESSLIST statement provides process information by collecting thread data from all active threads. The performance_schema_show_processlist variable determines which SHOW PROCESSLIST implementation to use:

- The default implementation iterates across active threads from within the thread manager while holding a global mutex. This has negative performance consequences, particularly on busy systems.
- The alternative SHOW PROCESSLIST implementation is based on the Performance Schema processlist table. This implementation queries active thread data from the Performance Schema rather than the thread manager and does not require a mutex.

To enable the alternative implementation, enable the performance_schema_show_processlist system variable. To ensure that the default and alternative implementations yield the same information, certain configuration requirements must be met; see Section 10.21.7, “The processlist Table”.

- performance_schema_users_size

Command-Line Format	--performance-schema-users-size=#
System Variable	performance_schema_users_size
Scope	Global
Dynamic	No

SET_VAR Hint Applies	No
Type	Integer
Default Value	-1 (signifies autoscaling; do not assign this literal value)
Minimum Value	-1 (signifies autoscaling; do not assign this literal value)
Maximum Value	1048576

The number of rows in the users table. If this variable is 0, the Performance Schema does not maintain connection statistics in the users table or status variable information in the status_by_user table.

Chapter 13 Performance Schema Status Variables

The Performance Schema implements several status variables that provide information about instrumentation that could not be loaded or created due to memory constraints:

```
mysql> SHOW STATUS LIKE 'perf%';
+-------------------------------------------+-------+
| Variable_name                             | Value |
+-------------------------------------------+-------+
| Performance_schema_accounts_lost          | 0     |
| Performance_schema_cond_classes_lost      | 0     |
| Performance_schema_cond_instances_lost    | 0     |
| Performance_schema_file_classes_lost      | 0     |
| Performance_schema_file_handles_lost      | 0     |
| Performance_schema_file_instances_lost    | 0     |
| Performance_schema_hosts_lost             | 0     |
| Performance_schema_locker_lost            | 0     |
| Performance_schema_mutex_classes_lost     | 0     |
| Performance_schema_mutex_instances_lost   | 0     |
| Performance_schema_rwlock_classes_lost    | 0     |
| Performance_schema_rwlock_instances_lost  | 0     |
| Performance_schema_socket_classes_lost    | 0     |
| Performance_schema_socket_instances_lost  | 0     |
| Performance_schema_stage_classes_lost     | 0     |
| Performance_schema_statement_classes_lost | 0     |
| Performance_schema_table_handles_lost     | 0     |
| Performance_schema_table_instances_lost   | 0     |
| Performance_schema_thread_classes_lost    | 0     |
| Performance_schema_thread_instances_lost  | 0     |
| Performance_schema_users_lost             | 0     |
+-------------------------------------------+-------+
```

For information on using these variables to check Performance Schema status, see Chapter 8, *Performance Schema Status Monitoring*.

Performance Schema status variables have the following meanings:

- `Performance_schema_accounts_lost`

  The number of times a row could not be added to the `accounts` table because it was full.

- `Performance_schema_cond_classes_lost`

  How many condition instruments could not be loaded.

- `Performance_schema_cond_instances_lost`

  How many condition instrument instances could not be created.

- `Performance_schema_digest_lost`

  The number of digest instances that could not be instrumented in the `events_statements_summary_by_digest` table. This can be nonzero if the value of `performance_schema_digests_size` is too small.

- `Performance_schema_file_classes_lost`

  How many file instruments could not be loaded.

- `Performance_schema_file_handles_lost`

  How many file instrument instances could not be opened.

- `Performance_schema_file_instances_lost`

  How many file instrument instances could not be created.

- `Performance_schema_hosts_lost`

  The number of times a row could not be added to the `hosts` table because it was full.

- `Performance_schema_index_stat_lost`

  The number of indexes for which statistics were lost. This can be nonzero if the value of `performance_schema_max_index_stat` is too small.

- `Performance_schema_locker_lost`

  How many events are “lost” or not recorded, due to the following conditions:

  - Events are recursive (for example, waiting for A caused a wait on B, which caused a wait on C).
  - The depth of the nested events stack is greater than the limit imposed by the implementation.

  Events recorded by the Performance Schema are not recursive, so this variable should always be 0.

- `Performance_schema_memory_classes_lost`

  The number of times a memory instrument could not be loaded.

- `Performance_schema_metadata_lock_lost`

  The number of metadata locks that could not be instrumented in the `metadata_locks` table. This can be nonzero if the value of `performance_schema_max_metadata_locks` is too small.

- `Performance_schema_mutex_classes_lost`

  How many mutex instruments could not be loaded.

- `Performance_schema_mutex_instances_lost`

  How many mutex instrument instances could not be created.

- `Performance_schema_nested_statement_lost`

  The number of stored program statements for which statistics were lost. This can be nonzero if the value of `performance_schema_max_statement_stack` is too small.

- `Performance_schema_prepared_statements_lost`

  The number of prepared statements that could not be instrumented in the `prepared_statements_instances` table. This can be nonzero if the value of `performance_schema_max_prepared_statements_instances` is too small.

- `Performance_schema_program_lost`

  The number of stored programs for which statistics were lost. This can be nonzero if the value of `performance_schema_max_program_instances` is too small.

- `Performance_schema_rwlock_classes_lost`

  How many rwlock instruments could not be loaded.

- `Performance_schema_rwlock_instances_lost`

  How many rwlock instrument instances could not be created.

- `Performance_schema_session_connect_attrs_longest_seen`

  In addition to the connection attribute size-limit check performed by the Performance Schema against the value of the `performance_schema_session_connect_attrs_size` system variable, the server performs a preliminary check, imposing a limit of 64KB on the aggregate size of connection attribute data it accepts. If a client attempts to send more than 64KB of attribute data, the server rejects the connection. Otherwise, the server considers the attribute buffer valid and tracks the size of the longest such buffer in the `Performance_schema_session_connect_attrs_longest_seen` status variable. If this value is larger than `performance_schema_session_connect_attrs_size`, DBAs may wish to increase the latter value, or, alternatively, investigate which clients are sending large amounts of attribute data.

  For more information about connection attributes, see Section 10.9, “Performance Schema Connection Attribute Tables”.

- `Performance_schema_session_connect_attrs_lost`

  The number of connections for which connection attribute truncation has occurred. For a given connection, if the client sends connection attribute key-value pairs for which the aggregate size is larger than the reserved storage permitted by the value of the `performance_schema_session_connect_attrs_size` system variable, the Performance Schema truncates the attribute data and increments `Performance_schema_session_connect_attrs_lost`. If this value is nonzero, you may wish to set `performance_schema_session_connect_attrs_size` to a larger value.

  For more information about connection attributes, see Section 10.9, “Performance Schema Connection Attribute Tables”.

- `Performance_schema_socket_classes_lost`

  How many socket instruments could not be loaded.

- `Performance_schema_socket_instances_lost`

  How many socket instrument instances could not be created.

- `Performance_schema_stage_classes_lost`

  How many stage instruments could not be loaded.

- `Performance_schema_statement_classes_lost`

  How many statement instruments could not be loaded.

- `Performance_schema_table_handles_lost`

  How many table instrument instances could not be opened. This can be nonzero if the value of `performance_schema_max_table_handles` is too small.

- `Performance_schema_table_instances_lost`

  How many table instrument instances could not be created.

- `Performance_schema_table_lock_stat_lost`

The number of tables for which lock statistics were lost. This can be nonzero if the value of `performance_schema_max_table_lock_stat` is too small.

- `Performance_schema_thread_classes_lost`

  How many thread instruments could not be loaded.

- `Performance_schema_thread_instances_lost`

  The number of thread instances that could not be instrumented in the `threads` table. This can be nonzero if the value of `performance_schema_max_thread_instances` is too small.

- `Performance_schema_users_lost`

  The number of times a row could not be added to the `users` table because it was full.

Chapter 14 Using the Performance Schema to Diagnose Problems

Table of Contents

The Performance Schema is a tool to help a DBA do performance tuning by taking real measurements instead of “wild guesses.” This section demonstrates some ways to use the Performance Schema for this purpose. The discussion here relies on the use of event filtering, which is described in Section 5.2, “Performance Schema Event Filtering”.

The following example provides one methodology that you can use to analyze a repeatable problem, such as investigating a performance bottleneck. To begin, you should have a repeatable use case where performance is deemed “too slow” and needs optimization, and you should enable all instrumentation (no pre-filtering at all).

1. Run the use case.
2. Using the Performance Schema tables, analyze the root cause of the performance problem. This analysis relies heavily on post-filtering.
3. For problem areas that are ruled out, disable the corresponding instruments. For example, if analysis shows that the issue is not related to file I/O in a particular storage engine, disable the file I/O instruments for that engine. Then truncate the history and summary tables to remove previously collected events.
4. Repeat the process at step 1.

   With each iteration, the Performance Schema output, particularly the `events_waits_history_long` table, contains less and less “noise” caused by nonsignificant instruments, and given that this table has a fixed size, contains more and more data relevant to the analysis of the problem at hand.

   With each iteration, investigation should lead closer and closer to the root cause of the problem, as the “signal/noise” ratio improves, making analysis easier.
5. Once a root cause of performance bottleneck is identified, take the appropriate corrective action, such as:

   - Tune the server parameters (cache sizes, memory, and so forth).
   - Tune a query by writing it differently,
   - Tune the database schema (tables, indexes, and so forth).
   - Tune the code (this applies to storage engine or server developers only).
6. Start again at step 1, to see the effects of the changes on performance.

The `mutex_instances.LOCKED_BY_THREAD_ID` and `rwlock_instances.WRITE_LOCKED_BY_THREAD_ID` columns are extremely important for investigating performance bottlenecks or deadlocks. This is made possible by Performance Schema instrumentation as follows:

1. Suppose that thread 1 is stuck waiting for a mutex.
2. You can determine what the thread is waiting for:

```
SELECT * FROM performance_schema.events_waits_current
WHERE THREAD_ID = thread_1;
```

   Say the query result identifies that the thread is waiting for mutex A, found in `events_waits_current.OBJECT_INSTANCE_BEGIN`.

3. You can determine which thread is holding mutex A:

```
SELECT * FROM performance_schema.mutex_instances
WHERE OBJECT_INSTANCE_BEGIN = mutex_A;
```

   Say the query result identifies that it is thread 2 holding mutex A, as found in `mutex_instances.LOCKED_BY_THREAD_ID`.

4. You can see what thread 2 is doing:

```
SELECT * FROM performance_schema.events_waits_current
WHERE THREAD_ID = thread_2;
```

14.1 Query Profiling Using Performance Schema

The following example demonstrates how to use Performance Schema statement events and stage events to retrieve data comparable to profiling information provided by `SHOW PROFILES` and `SHOW PROFILE` statements.

The `setup_actors` table can be used to limit the collection of historical events by host, user, or account to reduce runtime overhead and the amount of data collected in history tables. The first step of the example shows how to limit collection of historical events to a specific user.

Performance Schema displays event timer information in picoseconds (trillionths of a second) to normalize timing data to a standard unit. In the following example, `TIMER_WAIT` values are divided by 1000000000000 to show data in units of seconds. Values are also truncated to 6 decimal places to display data in the same format as `SHOW PROFILES` and `SHOW PROFILE` statements.

1. Limit the collection of historical events to the user that runs the query. By default, `setup_actors` is configured to allow monitoring and historical event collection for all foreground threads:

```
mysql> SELECT * FROM performance_schema.setup_actors;
+------+------+------+---------+---------+
| HOST | USER | ROLE | ENABLED | HISTORY |
+------+------+------+---------+---------+
| %    | %    | %    | YES     | YES     |
+------+------+------+---------+---------+
```

   Update the default row in the `setup_actors` table to disable historical event collection and monitoring for all foreground threads, and insert a new row that enables monitoring and historical event collection for the user that runs the query:

```
mysql> UPDATE performance_schema.setup_actors
       SET ENABLED = 'NO', HISTORY = 'NO'
       WHERE HOST = '%' AND USER = '%';
mysql> INSERT INTO performance_schema.setup_actors
       (HOST,USER,ROLE,ENABLED,HISTORY)
       VALUES('localhost','test_user','%','YES','YES');
```

   Data in the `setup_actors` table should now appear similar to the following:

```
mysql> SELECT * FROM performance_schema.setup_actors;
+-----------+-----------+------+---------+---------+
| HOST      | USER      | ROLE | ENABLED | HISTORY |
+-----------+-----------+------+---------+---------+
| %         | %         | %    | NO      | NO      |
| localhost | test_user | %    | YES     | YES     |
+-----------+-----------+------+---------+---------+
```

2. Ensure that statement and stage instrumentation is enabled by updating the `setup_instruments` table. Some instruments may already be enabled by default.

```
mysql> UPDATE performance_schema.setup_instruments
       SET ENABLED = 'YES', TIMED = 'YES'
       WHERE NAME LIKE '%statement/%';
mysql> UPDATE performance_schema.setup_instruments
       SET ENABLED = 'YES', TIMED = 'YES'
       WHERE NAME LIKE '%stage/%';
```

3. Ensure that `events_statements_*` and `events_stages_*` consumers are enabled. Some consumers may already be enabled by default.

```
mysql> UPDATE performance_schema.setup_consumers
       SET ENABLED = 'YES'
       WHERE NAME LIKE '%events_statements_%';
mysql> UPDATE performance_schema.setup_consumers
       SET ENABLED = 'YES'
       WHERE NAME LIKE '%events_stages_%';
```

4. Under the user account you are monitoring, run the statement that you want to profile. For example:

```
mysql> SELECT * FROM employees.employees WHERE emp_no = 10001;
+--------+------------+------------+-----------+--------+------------+
| emp_no | birth_date | first_name | last_name | gender | hire_date |
+--------+------------+------------+-----------+--------+------------+
|  10001 | 1953-09-02 | Georgi     | Facello   | M      | 1986-06-26 |
+--------+------------+------------+-----------+--------+------------+
```

5. Identify the `EVENT_ID` of the statement by querying the `events_statements_history_long` table. This step is similar to running `SHOW PROFILES` to identify the `Query_ID`. The following query produces output similar to `SHOW PROFILES`:

```
mysql> SELECT EVENT_ID, TRUNCATE(TIMER_WAIT/1000000000000,6) as Duration, SQL_TEXT
       FROM performance_schema.events_statements_history_long WHERE SQL_TEXT like '%10001%';
+----------+----------+-----------------------------------------------------------+
| event_id | duration | sql_text                                                  |
+----------+----------+-----------------------------------------------------------+
|       31 | 0.028310 | SELECT * FROM employees.employees WHERE emp_no = 10001 |
+----------+----------+-----------------------------------------------------------+
```

6. Query the `events_stages_history_long` table to retrieve the statement's stage events. Stages are linked to statements using event nesting. Each stage event record has a `NESTING_EVENT_ID` column that contains the `EVENT_ID` of the parent statement.

```
mysql> SELECT event_name AS Stage, TRUNCATE(TIMER_WAIT/1000000000000,6) AS Duration
       FROM performance_schema.events_stages_history_long WHERE NESTING_EVENT_ID=31;
+------------------------------+----------+
| Stage                        | Duration |
+------------------------------+----------+
| stage/sql/starting           | 0.000080 |
| stage/sql/checking permissions | 0.000005 |
| stage/sql/Opening tables     | 0.027759 |
| stage/sql/init               | 0.000052 |
| stage/sql/System lock        | 0.000009 |
| stage/sql/optimizing         | 0.000006 |
```

```
| stage/sql/statistics           | 0.000082 |
| stage/sql/preparing            | 0.000008 |
| stage/sql/executing            | 0.000000 |
| stage/sql/Sending data         | 0.000017 |
| stage/sql/end                  | 0.000001 |
| stage/sql/query end            | 0.000004 |
| stage/sql/closing tables       | 0.000006 |
| stage/sql/freeing items        | 0.000272 |
| stage/sql/cleaning up          | 0.000001 |
+--------------------------------+----------+
```

14.2 Obtaining Parent Event Information

The `data_locks` table shows data locks held and requested. Rows of this table have a `THREAD_ID` column indicating the thread ID of the session that owns the lock, and an `EVENT_ID` column indicating the Performance Schema event that caused the lock. Tuples of (`THREAD_ID`, `EVENT_ID`) values implicitly identify a parent event in other Performance Schema tables:

- The parent wait event in the `events_waits_xxx` tables
- The parent stage event in the `events_stages_xxx` tables
- The parent statement event in the `events_statements_xxx` tables
- The parent transaction event in the `events_transactions_current` table

To obtain details about the parent event, join the `THREAD_ID` and `EVENT_ID` columns with the columns of like name in the appropriate parent event table. The relation is based on a nested set data model, so the join has several clauses. Given parent and child tables represented by `parent` and `child`, respectively, the join looks like this:

```
WHERE
  parent.THREAD_ID = child.THREAD_ID        /* 1 */
  AND parent.EVENT_ID < child.EVENT_ID      /* 2 */
  AND (
    child.EVENT_ID <= parent.END_EVENT_ID   /* 3a */
    OR parent.END_EVENT_ID IS NULL          /* 3b */
  )
```

The conditions for the join are:

1. The parent and child events are in the same thread.
2. The child event begins after the parent event, so its `EVENT_ID` value is greater than that of the parent.
3. The parent event has either completed or is still running.

To find lock information, `data_locks` is the table containing child events.

The `data_locks` table shows only existing locks, so these considerations apply regarding which table contains the parent event:

- For transactions, the only choice is `events_transactions_current`. If a transaction is completed, it may be in the transaction history tables, but the locks are gone already.
- For statements, it all depends on whether the statement that took a lock is a statement in a transaction that has already completed (use `events_statements_history`) or the statement is still running (use `events_statements_current`).
- For stages, the logic is similar to that for statements; use `events_stages_history` or `events_stages_current`.

- For waits, the logic is similar to that for statements; use `events_waits_history` or `events_waits_current`. However, so many waits are recorded that the wait that caused a lock is most likely gone from the history tables already.

Wait, stage, and statement events disappear quickly from the history. If a statement that executed a long time ago took a lock but is in a still-open transaction, it might not be possible to find the statement, but it is possible to find the transaction.

This is why the nested set data model works better for locating parent events. Following links in a parent/child relationship (data lock -> parent wait -> parent stage -> parent transaction) does not work well when intermediate nodes are already gone from the history tables.

The following scenario illustrates how to find the parent transaction of a statement in which a lock was taken:

Session A:

```
[1] START TRANSACTION;
[2] SELECT * FROM t1 WHERE pk = 1;
[3] SELECT 'Hello, world';
```

Session B:

```
SELECT ...
FROM performance_schema.events_transactions_current AS parent
  INNER JOIN performance_schema.data_locks AS child
WHERE
  parent.THREAD_ID = child.THREAD_ID
  AND parent.EVENT_ID < child.EVENT_ID
  AND (
    child.EVENT_ID <= parent.END_EVENT_ID
    OR parent.END_EVENT_ID IS NULL
  );
```

The query for session B should show statement [2] as owning a data lock on the record with `pk=1`.

If session A executes more statements, [2] fades out of the history table.

The query should show the transaction that started in [1], regardless of how many statements, stages, or waits were executed.

To see more data, you can also use the `events_xxx_history_long` tables, except for transactions, assuming no other query runs in the server (so that history is preserved).